# Learning Through Examples : The Symbolic Integration Problem

*By*

**DESHPANDE SUDHEER RAGHUNATH**

**DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING**

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

**APRIL, 1993**

# Learning Through Examples : The Symbolic Integration Problem

*A Thesis Submitted*
*in Partial Fulfilment of the Requirements*
*for the Degree of*

MASTER OF TECHNOLOGY

*by*

DESHPANDE SUDHEER RAGHUNATH

to the

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

April, 1993

# CERTIFICATE

This is to certify that the work contained in the thesis titled, **LEARNING THROUGH EXAMPLES: THE SYMBOLIC INTEGRATION PROBLEM**, was carried out under my supervision by **DESHPANDE SUDHEER RAGHUNATH** and it has not been submitted elsewhere for a degree.

[illegible] 22/4

**Prof. R. M. K. SINHA**
Professor and Head
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur

CSE- 1993-M-RAG - LEA

*Dedicated to*

My Parents

# Acknowledgements

I feel very light after coming to this wonderful and happy juncture, where I can express my feelings of gratitude towards many people who have made ME the Person, as I stand here.

Let me first thank Prof. R.M.K. Sinha, my thesis advisor, for teaching me how a person should proceed when he wants to do some research. I am really thankful to him for giving me a rare combination of freedom and guidance for defining my problem. Discussions with him were always very fruitful and explorative. He was very understanding and provided personal advices in time of distress. Also I would like to express my sincerest gratitude towards all my teachers.

On the personal front, there are n-number of friends, relatives and well-wishers whom I owe so much. All my friends are very jolly, enthusiastic, and energetic, providing full encouragement, support and confidence in all matters. Harshad, Navin, Nitin, Natava, Dev, Alexei, Atul and Ashish formed a very nice group. Rahul, Anya, Tatya, Wadya were another set. My friends at Pune, Sangli and Solapur also deserve some credit. And yes, how can I forget Capt. (Now Major) Bhat, who was like an elder brother to me? I'll always cherish the love and affection provided by the AVAM's.

I know, I am here just because of my brother and sister. My parents always provided the best environment possible for me.

At the end I'd like to say, the woman behind the successful man always telling him that he is wrong, yes, Chinmoyee, was the inspiration and motivation for what I am.

SUDHEER

# ABSTRACT

Any intelligent system should be able to improve its performance through experience. The problem solving systems should use their experience and solve new examples from the same class more efficiently. These systems can use the traces of earlier examples to find these similarities. Solving an integration problem is a very good example of problem solving. Since integration is the reverse process of derivation, there are no definite solutions to an integral.

The problem solving performance can be improved in many ways using different techniques. Adding a new operator or a macro-operator, changing the preconditions of an operator, improving the language used to define the states, operators and preconditions, or improving the processes used for operator selection, matching or firing can be termed as Learning. If the operator search process is guided by some theory, then reorganization or aquisition of new concepts for this theory may improve the overall system performance. This learning is achieved by observing the solution trace of some solved example and then generalizing concepts from it.

The idea is to generate some heuristics after looking at the trace of a single example and then taking help from the domain theory. This system first tries to solve the example provided by the tutor, and then generates some selection heuristics, some rejection heuristics and some preference heuristics alongwith some possible macro-ops using the trace of that example. The system uses constructive induction for selection heuristics. Thus, this approach collects benefits from both the main streams, the empirical and the explanation based learning approaches.

*" Thought without Learning is useless,*

*Learning without Thought is dangerous "*

*- Confucious*

# Contents

# List of Figures

# Chapter 1

# Introduction

*Machine Learning* has been dominating the AI scene from its very begining. Today there are many active research projects spanning a whole range of Machine Learning methods. several focusing on the theory of learning and others on improving problem solving performance in complex domains.

## 1.1 Why Learning?

The obscurity of the genetically endowed abilities in a biological system has fascinated researchers from various fields of biology, psychology, philosophy and AI alike. The quest for a cognitive invariant in humans brings forth a clear candidate that is *Learning*, which denotes the innate ability to acquire facts, skills, and more abstract concepts. This ability to learn, to adapt, to modify behaviour is an inseperable component of human intelligence.

When we term a person as a intelligent being we are actually referring to his abilities with respect to analyzing, adapting, experimenting and solving the real world problems. For all this one needs to use a large knowledge base, but only knowledge base is not enough: the ability to acquire new knowledge and to refine old alongwith an effective, efficient organization of this large chunk with opportunistic retrieval is important. An underlying assumption of many Machine Learning researchers is that learning is a prerequisite to any form of true intelligence.

Therefore understanding Human Learning well enough to reproduce aspects of the learning behaviour in Computer systems is in itself a worthy scientific goal. This has many applications like apart from making the smarter systems, it will help in developing better educational tchniques for human beings itself. An equally basic scientific objective is the exploration of alternative learning mechanisms. including the discovery of different induction methods. the scope and limitation of certain methods, the infomation that must be available to the learner. etc. Most theoretical work in Machine Leaarning has centered around the creation . characterization and analysis of general learning methods. with the major emphasis on analyzing generality and performance.

## 1.2 What is Learning?

When we say learning we are really addressing a very large spectrum of activities. like we say that we learn (memorize) multiplication tables. we learn (acquire motor skills for) cycling, we learn (analyze and synthesize) languages, etc. The question arises what is exactly meant by learning? According to [Sim83] "*Learning denotcs changes in the system that are adaptive in the sense that they enable the system to do thc same task or tasks drawn from the same population more efficiently and more effectively the next time.*" which is a well accepted definition. It can be very well seen that if a system is not able to improve its performance over experience. we can't call it a truly intelligent system.

Machine Learning like knowledge representation and search techniques. cuts across all problem areas of AI: problem solving, theorem proving, analogical and nonmonotonic reasoning, game playing. pattern recognition, NLP, vision. robotics, planning, expert systems. and so on. In principle, progress in Machine Learning can benefit all these areas; it is truly at the core of AI.

## 1.3 Taxonomy of Machine Learning Methods

As stated earlier the learning mechanism addresses a wide range of activities spanning many different domains, and hence a number of different methods have been evolved to learn these different types of activities. As we have seen any change in the system that improves the system performance can be termed as learning, we have inherently different techniques to bring these different types of changes into effect. Typically, any AI system has two parts: knowledge base and reasoning mechanism; so all the methods that can enhance or improve the knowledge base or make changes in reasoning mechanism can be termed as learning methods.

Here we provide two different kinds of classification of these methods:

- Classification based on Underlying Learning Strategy.
- Classification based on Paradigms.

### 1.3.1 Classification based on Underlying Learning Strategy

The following are the different learning strategies:

1. **Rote Learning** : In this method no inference or other transformation on knowledge base is required on the part of the learner.
2. **Learning from Instruction** : Acquiring knowledge from a teacher[1] requiring that learner transform the knowledge in order to reorganize the earlier existing knowledge base. Still in this case the burden to organize the input knowledge and feed it to the system is on the teacher only.
3. **Learning by Analogy** : Acquiring new facts by transforming and augmenting existing knowledge base that depicts strong similarity with the new concept into a form effectively useful in the new situation. This requires more inferencing from learner's part. A fact or skill analogous in relevant parametters must be retrieved from the memory; then the retrieved K must be transformed, applied to the new situation and stored for future use.
4. **Learning from Examples** : Given a set of examples and counter-examples of a concept, the learner induces a general concept description that describes all the positive examples and none of the counter-examples. Here the source of examples may be either a teacher, or the learner itself or the external environment and there are two possibilities that are either only positive examples available or both positive as well as negative examples available. This method is slightly complex in the sense that the system doesn't have a seed concept as it exists in above method.
5. **Learning from observation and Discovery** : This is a very general form of inductive learning that includes discovery systems, teory formation tasks, the creation of a classification criterion, etc. This is also termed as unsupervised learning. In this case the learner is not provided with any set of instances of a particular concept,

[1]A teacher can be a book, a person coding knowledge or any other organised knowledge source

nor it is told about the classification of instances into positive, negative, etc. The learner has to take all these decisions including whether to form a single concept or multiple concepts to encapture all the examples. In this case the learner may do passive observation and classify the instances, or it can actively experiment with the environment and use the generate and test hypothesis on its own.

### 1.3.2 Classification based on Paradigms

The following are the four major Machine Learning paradigms:

1. **The Induction Paradigm :**

   The most widely studied method for symbolic learning is inducing a generalised desription about a concept in the light of some examples of (may be presumably) that concept. The design space of an inductive system is determined by many important dimensions like :

   - *Description Language* for the input instances and output concepts.
   - *Noise and instance classification.*
   - *Concept Type* denoting characteristic or descriminative concepts.
   - *Source of instances.*
   - *Incremental or one-shot induction.*

2. **The Analytic paradigm :**

   This is based on analytic learning from few examples (often a single one) plus a rich underlying domain theory[2]. The methods involved are more deductive in nature, utilising past problem solving experience to guide when solving new examples. Analytic methods focus on improving system performance than increasing the concept library. Presently, analytic methods focus on explanation based learning, multi-level chunking, macro-operator formation and derivational analogy. The fundamental issues in these analytic methods are :

   - *Representation of instances.*
   - *Learning from success and failure.*

[2] Domain Theory consists of expert K about the domain and ideally, it should be available as a plug-in module.

  - *Degree of generalization.*
  - *Closed Vs Open loop learning.*

3. **The Genetic paradigm :**

   The genetic algorithms are inspired by a direct analogy to mutations in biological reproduction (cross-overs and point mutations) and Darwinian natural selection (survival of the fittest). Variants of a concept description are the individual species, and induced changes and recombinations (cross-overs and mutations) are tested against an objective function (function governing criterion for natural slection) to see which one to preserve in the gene pool. After a fixed number of generations the best concept descriptors are selected. In principle, genetic algorithms encode a parallel search through concept space, with each process attempting coarse-grain hill climbing.

4. **The Connectionist paradigm**

   Also called *Neural Networks* or *Parallel Distributed Systems*, the connectionist learning systems have removed the limitations of perceptrons by introducing hidden layers. These can be classified in two manners: those that use distributed representations - where a concept corresponds to an activation pattern, potentially the entire network - and those that use localised representations where physical portions of the network correspeond to individual concepts. These systems learn to discriminate among equivalent classes of patterns from an input domain in a holistic manner. These are presented with the representative instances of each class, correctly labelled, and they learn to recognise these and oher instances of each class. Learning consists of adjusting the weights in a fixed topology network using some learning algorithm.

## 1.4 The Problem

Symbolic Integration, just like chess, has always attracted the attention of AI researchers since it provides a good example of problem solving. In general any problem solving example consists of following parts: A set of states that the problem may achieve, a set of operators that can get applied to these states, a language to define these states and operators, and a mechanism to apply these operators and to know when the solution is achieved.

Although simple problem solving means take the initial state. go on applying the operators till you reach the final state. But human beings don't solve problems like this. they do take advantage of their earlier experience to decide which operators to apply and when. The learning for problem solving comes into picture at this point. Human beings form heuristics to apply operators, assign priorities among the operators. form macro-operators. and generalize plans of problem solving. They are also found to experiment with the problem situations in order to test their own hypothesis. in fact problem solving provides a natural platform to use learning mechanisms.

Now, we can use different approaches to learn about problem solving: we can use the inductive paradigm, or the genetic paradigm or the analytic paradigm. Also. we can have two alternatives for learning: using a seqeunce of examples or using a single example to generate some heuristics and macro-operators.

Many researchers have already tried to build systems for symbolic integration. Joal Moses at MIT has developed a very good system MACSYMA which solves almost all integrals and makes use of Integration by partial Fractions mathod but it's not a learning system. One more celebrated example is LEX system explained in [Mit83]. LEX tries to learn heuristics by experimenting on its own. It forms some hypothesis heuristics and then generates test integrals. solves them. and after the analysis of the solution, it tries to refine the heuristics.

We will see how we can learn the heuristics and macro-operators from a single example. We will consider making heuristics and forming macro-operators for Integration using By Parts rule and Integration by Substitution.

## 1.5 Thesis Organization

This thesis report contains following chapters:

Chapter 2 presents a survey on various Machine Learning methods and meant to give the reader a brief introduction to various learning methods.

Chapter 3 contains the design of the system which outlines the specifications of the system for learning.

Chapter 4 gives the implementation details about the Symbolic Integration Module while Chapter 5 gives the implementation details about the Learner which tries to form the heuristics and the macro-ops.

Chapter 6 concludes the efforts with a discussion about the performance of the system.

The appendices contain the class definitions, rules, complexity analysis, and some sample output files.

# Chapter 2

# A Survey

In this chapter, we are going to cover Inductive Learning, Explanation Based Learning, genetic Algorithms, and Problem Reformulation. Although, in literature, one can find numerous papers on each of these topics seperately, efforts are being made to put together the good points of all these approaches, taking into account the complexity of real world, and how human being is able to cope up with it, leading to the unification of these approaches.

## 2.1 Inductive Learning

Inductive Learning being most studied of all learning methods, is covered in much breadth and depth in the literature which covers the framework, analysis and applications about it.

### 2.1.1 Theory of Inductive Learning

Inductive LearninInductive Learning is a process of acquiring knowledge through inductive inferences from teacher provided facts. Therefore, Inductive Learning has got an inherent weakness that the acquired knowledge can not be validated. As deductive inference preserves truth value, inductive inferencing preserves falsity. If H — F is valid, and H is true then by modus ponuns, we can infer F (deduction, truth preserving) but if F is true and then deriving H from F id falsity prserving (induction).

1. **Types of Inductive Learning** Following are some of the differentiating points between specifying types of Inductive Learning.

- *Concept acquisition Vs Descriptive generalization*: We have already seen about these two major types, learning from examples (concept acquisition) and learning from observation (descriptive generalisation). In concept acquisition, the observational statements are characterizations of some objects preclassified by the teacher into one or more classes (concepts) and in descriptive generalization the goal is to determine a general description (a law, a theory) characterizing a colloction of observations. In concept acquisition, we learn either a *characteristic descroption* or a *discriminant description* or a *sequence extrapolation rule*. And in descriptive generalisation, we either formulate a theory, discovering some patterns in observational data, or determining a taxonomic description of a collection of objects.
- *Characteristic vs Descriminant description*: A characteristic desription is an expression that satisfies the completeness condition in the sense that it desribes which all instances can be there in that particular concept and a descriminant description is an expression that satisfies the consistancy condition in the sense that it descriminates between instances of different concepts.
- *Single Vs Multiple concept Learning*: The system can focus its attention on only one concept or if it is insufficient to classify the instances, it can consider multiple concept sceneraio to describe the instances at hand.

2. **Description Language:** One of difficulites with Inductive Learning is open-endedness This means that when one makes an inductive assertion about some aspect of reality there is no natural limit to the level of detail in which this reality may be described, or to the richness of forms in which this assertion can be expressed. consequently, while conducting research in this area, it is necessary to circumscribe very carefully the goals and the problem to be solved. This includes the definition of the language to be chosen for description.

3. **Problem background knowlwdge:** As we have seen one may construct infinitely many inductive assertions, the *problem backbground knowledge* also called domain knowledge is necessary to constrain the space of possible inductive assertions and locate the most desirable ones. X spefies this as the bias for inductive learning,

defining *bias* as any sort of knowledge used to prefer one inductive assertion over the other and he specifies the need for such a bias. The domain knowledge has got following components : Information about descriptors, assumption about the form of observational and inductive assertions, a preference criterion for properties of inductive assertions, and a variety of rules, heuristics, and other problem dependent knowledge.

- Relevance of initial descriptors: If the initial descriptors are completely relevant, the task of the learning system may be just to relate these descriptor in a proper manner. If the initial descriptors are indirectly relevant, the system has to find out the relevanc through constructive induction and if the inittial descriptors are partially relevant then the system has to learn inductive assertions selectively choosing relevant and sufficient assertions.
- Annotation of descriptor: An annotation of the descriptor is a store of background information about this descriptor tailored to the learning problem under consideration.
- Domain and type of a descriptor: It specifies the structure of the domain.
- Constraints on the description space: The constraints on the acceptable concept description are due to interdependence among values, properties of the descriptors and interrelationships among descriptors.
- The domain knowledge specifies the form of observational and inductive assertions also alongwith the preference criterion to select the inductive assertion from the total available space.

4. **Generalization Rules:** A generalization rule is a transformation of a description into a more general description, one that tautologically implies tha initial description and the specialization rule is just opposite of it specifying the logical consequence of a description while the reformulation rule tranforms a description into a logically-equivalent one.

   Following are the some of the select generalization rules:

   - *Dropping condition rule* in which a condition may be removed from the conjunctive precedent part of a description.

- *Adding alternative rule* in which a condition may be added disjunctively to the precedent part of a description.
- *Extending reference rule* in which a description may be extended to another member of the same domain.
- *Closing interval rule.*
- *Climbing generalization tree rule.*
- *Turning constraints into variables rule.*
- *Turning conjunction into disjunction rule.*
- *Inductive resolution rule.*

5. **Constructive Induction:** In constructive induction we generate new descriptors that are not present in the original observational statements. It makes use of the knowledge about the relationships between different concepts to arrive at new descriptors to be used in the statement.

   We can say that

   if $C\&F1 \Rightarrow K$ and $F1 \rightarrow F2$,

   then $C\&F2 \Rightarrow K$,

   denotes how relationship between facts F1 and F2 is made use of to arrive at a new inductive assertion and this is termed as constructive induction.

### 2.1.2 Comparative review of Learning from examples

Here [Die83] have examined metods for finding the maximally-specific conjunctive generalizations (MSCGs) that cover all of the trining examples of a given concept. The paper points out some important aspects of learninig from examples which are:

- Representation of inductive assertions and the operators.
- Type of description sought.
- Rules of generalization to be used for the transformation.
- Constructive induction.

- Control strategy for search through the description space.
- General vs problem oriented approach.t

The paper studies some data-driven and some model-driven methods using following evaluation criteria :

- Adequacy of the representation language.
- Rules of generalization implemented.
- Computational efficiency.
- Flexibility and extensibility.

### 2.1.3 Generalization as Search

The process of learning is nothing but generalization; from examples to concepts. In the paper [Mit90a] has described how the generalization process can be looked as the search process for appropriate inductive assertions from the total space of possible assertions. He has classified the search methods into two strategies: Data driven and generate-and-test. The strategies like depth first search. breadth first search and version space strategies are classified as data-driven strategies and generate-and-test strategies are from general to specific and from specific to general. The paper identifies some issues relating the generalization language and a method to represent the partially learned generalizations and some method to handle the inconsistancies in the descriptions of the assertions.

### 2.1.4 Learning by Analogy

In the paper [Car83] has given a good account of the method of learning by analogy. Analogical learning is a powerful mechhanism to exploit past experience in planning and problem solving. In the analogical inferencing the crucial part is to find out that the new concept is closer to some earlier defined concept, i.e. the *familiarity* of the problem space. He has used the difference function in the *Means-and-Ends Analysis* as the similarity metric. In analogical problem-solving the first phase accounts for the remembrance of some earlier concept and then the second phase transforms the old concept into the new one. He has defined some transformation operators to tranform the analogical space. He has also addressed the problem of lerarning generalized solution procedures to form seed concepts to be used afterwards.

### 2.1.5 The Need for Biases in Inductive Learning

As mentioned earlier Inductive Learning also requires some form of knowldge to make 'good' inductive assertions. Any basis for choosing one generalization over another one, other than strict consistancy with the observed training instances is called bias [Mit90b]. For a generalization system to be unbiased, it has to consider all possible generalization of an example concept. We can now easily see that such a system is useless, since for such a system any new instance will match every generalization consistant with observed instances if and only if it is identical to one of the observed instances.

The useful classes of biases can be:

- *Factual knowledge of the domain.*
- *Intended use of learned generalization.*
- *Knowledge about the source of training data.*
- *Bias toward simplicity and generality.*
- *Analogy with previously learned generalizations.*

### 2.1.6 Limitations of Inductive Learning

The paper [Die89] explores the proposition that inductive learning from examples is fundamentally limited to learning only a small fraction of the total space of possible hypotheses. The paper finds the following limitations:

- There are no general purpose learning algorithms that can learn any concept.
- Different classes of learning problems call for different learning algorithms.
- Results suggest that human learning involves much more learning from the examples.
- There are certain upper bound on learning which means that any inductive algorithm will not be able to improve learning performance significantly.

- People are found to learn well from a wide variety of domains than the algorithms.

## 2.2 Explanation Based Learning

Explanation Based Learning (EBL) systems try to explain why a particular example is an instance of a concept. The explanations are then converted into operatinal recognition rules. Thus, the EBL approach is analytical and knowledge-intensive. Explanatory schema acquisition, constraint-based generalization, explanation based generalization are some of the examples of EBL. In constraint based generalization by computing the weakest preconditions of the seqeuence of rules, one can generate the explanation. The fundamental theme of the EBL methods is that there is a general target concept which is to be learned. The EBL learns by transforming the knowledge bias. The example plays a very important role to guide the search for an operational description.

### 2.2.1 Current Issues in EBL

- *Formalizing EBL.*
- *Domain theory*: may be imperfect and may be intractable, or incomplete or inconsistant.
- *The expalanation.*
- *The example.*
- *The operationality criterion.*
- *The target concept.*

### 2.2.2 Explanation Based Generalization

In EBG the problem situation is as follows:

**Given:**

- *Goal Concept*: A concept definition describing the concept to be learned.
- *Training Example*: An example of the goal concept.

- *Domain Theory*: A set of rules and facts to be used in expaining hoe the trining example is an example of the goal concept.

- *Operationality Criterion*: A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed.

**Determine:** A ganeralization of the training example that is sufficient concept definition for the goal concept and that satisfies the operationality criterion.

Thus, EBG produces a justified generalization from a single training example. [Mit90c] proposes an alternative method for combining the results of EBGs from multiple training examples.

### 2.2.3 Explanation Based Learning

[DeJ90] have criticised the EBG and have defined a broader term EBL. The paper denotes many points such as the EBG specification does not specify how the explanation is generated; the operationality criterion in EBG is in itself not an operational one, etc. They have come across two problems with the definition of operationality criterion and they are: First, the operationality of the operationality criterion itself which assumes the definition of some predicate and second, the specification of the predicate alone will not insure ease of evaluation. The difficulty of discovering a truth value is assocaited with the proposition as a whole and not just the predicate.

They have also pointed out one technical problem with EBG algorithm that is the EBG fails to mention the necessity of an additional step when generalizing an expanation and that step is starting from the generalized antecedents, rederive the proof to obtain the generalized goal concept. They have provided an alternative specification of the EBG process alongwith explicitly mentioning the generalization procedure.

### 2.2.4 EBL for Problem Solving

The PRODIGY[Min89], a domain-independent problem solver, has a learning component which utilises EBL to explain new concepts alongwith some other learning methods. The PRODIGY learns from successful, fialed as well as the nodes providing multiple choices. It forms rules for selection, preference and rejection of operators.

1. **The EBL component**: The EBL component specifies and explains the steps in the problem solver.
   - *Target concept coverage*: The system has four types of target concepts: succeeds, fails, sole-altenative, goal-interferrence and depending upon these target concepts, EBL forms preference, selection and rejection rules.
   - *Scope of the Theory*: domain theory has got architecture level axioms as well as domain level axioms.
   - *The mapping method*: mapping from problem trace into an explanation involves first selecting an example from the space tree and then constructing an explanation of that particular example. A target concept is specialized by retrieving an axiom that describes the concept and recursively specializing the axiom. the recursion terminates on encountering primitive formulas. The sequence of specialization proves that the example is a valid instance of the target concept.
   - *Operationality Criterion*: The learned control knowledge should not just be usable, but useful as well. The operationality criterion also includes utility check.
2. **Learning from Success**: To explain why a particular node succeeded, the EBL takes the full path and then start from backwards from the last operator giving solution. It can be easily seen how the last operator got success and for intermediate operators if we can show that after the applicatopn of that operator the problem is solvable, then we can declare that operator as successful also. In this way, working backwards, the EBL proves how a particular operator at a particular node became successful. PRODIGY learns Selection rules and preference rules from successful nodes. This method of learning is very similar to macro-operator formation process.
3. **Learning from Failure**: Similarly, to explain how a node failed, the path to the downmost failed child is considered and an explanation is generated to explain how the failure of that node is carried upwards in the tree to make the node in question a failure. This way PRODIGY learns the rejection rules.
4. **Learning from Goal Interferrence**: Goal interactions are found everywhere in planning. When we get such a goal interferrence, we need to determine which one to choose in order to get a better path. EBL tries to search for a better alternative

depending upon the increase in complexity of the problem state. The preference rules are generated using such goal interference.

### 2.2.5 Unifying themes for Inductive Learning and EBL

There are many papers in literature which study the unification of different types of learning methods especially, Inductive Learning and EBL. [Lan89] has shown how the two methods are closer than expected in many ways like Inductive Learning also requires domain knowledge to get better inductions, Inductive Learning also learns some bit from each example and EBL also can make use of multiple examples rather than a single example.

[Fla90] has shown how explantion based methods can be used in association with Inductive Learning to generalize the knowledge gained by individual example using EBL and then using Inductive Learning how we can combine them giving an entirely new concepts, which he has termed as Induction Over the Explained (IOE). Some other papers also show how multiple examples can be used to form concepts using the best of both of these methods.

## 2.3 genetic Algorithms

In genetic Algorithms (GA)[Boo89], the descriptors are stored in a 'gene store'. GA work on the principles of biological system. GA are generally used in learning from observation, in which the system has to classify a certain number of instances where the exact nimber os classes are not known and the instances are also not labelled. The GA can form some characteristic description for these classes.

### 2.3.1 The Algorithm

The algorithm is very simple as given below:

1. From the set of gene store, select pairs according to strength - the stronger the gene (descriptor), the more likely its slection.
2. Apply genetic operators to the pairs, creating "offspring" descriptors. The main genetic operators are mutation, cross-over, etc.
3. Replace the weakest descriptors with the offspring.

Therefore, it can be seen that apart from the main algorithm, the important thing is how one defines the strength of each descriptor.

### 2.3.2 Comparison with other mmethods

In terms of the other weak methods, GA can be thought of as a complex hierarchical generate-and-test process. The generator produces building blocks which are combined into complete objects. The differences between GA and other weak methods is the search for useful rules. The familiar weak methods focus on managing the complexity of the search space, emphasizing ways to avoid computationally prohibitive searches while GA proceed by managing the uncertainty of the search space. GA handles noise effortlessly because uncertainty reduction is at the heart of the procedure, but the price paid by GAs is that it robustly samples the space without concern for the difficulty of the problem; it can not use an obvious path to a solution. One difficulty with GA is that there is a possibility that the whole subsets of good rules are lost, but it is very rare.

## 2.4 Problem Reformulation

Research in the field of problem Solving, Expert Systems, and Machine Learning has been converging on the isue of problem representation. A system's ability to solve problems and acquire knowledge depends upon the initial problem representation. One solution to this bottleneck is to automatically alter the problem representation to another representation which is mmore efficient with respect to a given problem solving method and a given problem class.

### 2.4.1 Automating problem reformulation

This change of problem representation is called problem reformulation. The major representational changes are brought by defining Problem reduction operators and forming macro-operators. The SOJOURNER[Rid90] tries to derive critical state component first, then finding the invariants, then decomposing the solution path and finding inter-reduction and intra-reduction generalizations to achieve the problem reduction operators. To get macro-operators, it tries to get repetitions of opearators or operator sequences, thereby getting the macro-oprator which defines these repeatitions.

In general, the techniques for problem reformulation are :

- Compiling constraints.
- Removing irrelevant information.
- Removing redundant information.
- Deriving macro-operators.
- Deriving problem reduction schemas.
- Deriving iterative macro-operators.

### 2.4.2 Justified Reformulations

[Sub90] has analyzed the requirements for automating reformulations and showed the need for justifying shifts in conceptualization. She defines a notion called relevance among different concepts. The concept $B$ is said to be irrelevant to concept $A$ with respect to some theory $T$ if any changes in $B$ does not have a bearing on $A$ with respect to that theory $T$. In her doctoral thesis she has given a method to find out such irrelevance claims and transform the theory in a more compact form. Such a transforation will be justifiable under the irrelevance claim.

The domain theory for any learning system can be tested for such transformations which will help to an efficient reorganization of the theory. Again, there are two types of transformations: Deductive tranformations and inductive transformaions.

The deductive transformations make use of simple irrelevance claims where the goal representations are preserved through the change. But, the inductive reformulation makes it possible to define some goals which were not definable earlier. Utgoff has made use of inductive reformulation to change the bias (the grammer of expressions) so that the concept of even number was definable afterwards. He ha made use of it to define certain type of integrals which were not solvable earlier.

# Chapter 3

# Designing the System

The idea is to build a system that can learn some aspects of Symbolic Integration. Solving an integration problem is a very good example of problem solving. The intended system should solve such integration problems. Given, an example of a particular integrand, the system, should be able to use that example, in order to learn certain aspects about integration, highlightened by the example . The inherent complexity of the integration problem lies in the fact that it is the reverse process of derivation and there are no definite solutions to an integrand. All you have are some table lookups and some rules of thumb to guide your search. Therefore getting these rules of thumb (heuristics) is the primary task in order to solve an integral.

## 3.1 Problem Solving and Learning

The problem, in a problem solving domain is formulated as follows:

**Given:**

1. A set of states used to represent the status of the problem situation at a particular instant.
2. An initial state.
3. A set of final states.
4. A set of operators alongwith their preconditions.

**Goal:**

Given an instance of the initial state, reach the final state.

The procedure applied is as follows: Check whether any operator is applicable to that state or not, if yes, apply that oerator and get a new state. Continue this till you reach one of the final states.

The above defintion assumes many things as implicitly defined. They are the language used to define the states, operators and preconditions and the processes for operator selection, matching and firing. All these things do have a great influence on the problem solving performance of the system and hence, sometimes these things are explicitly defined. Any change in these parameters which will enhance the system performance can be termed as Learning. Adding a new operator or a macro-operator, changing the preconditions of an operator, improving the language used to define the states, operators and preconditions, or improving the processes used for operator selection, matching or firing can be termed as Learning. If the operator search process is guided by some theory, then reorganization or aquisition of new concepts for this theory may improve the overall system performance.

## 3.2 Problem Formulation

While formulating the problem, we have to first give a thought as to what aspects of the system should be adaptable to changes, the types of changes, and the manner in which the changes may be incorporated in the system. The changes may be automatically incorporated in the system or there may be a tutor involved in validating these changes before incorporating them.

There are two possibilities as far as a decision has to be made about when can the system learn. Either the system may learn while it is solving the example or it may learn the concepts at the end, when the example is fully solved. We would like the system to learn the things only at the end since that's the most fool-proof method, otherwise, in the first approach, one may have to unlearn some things should they prove useless afterwards in the solution space.

In our case, we would like the system to suggest some heuristics for the usage of the operators and to suggest some macro-operators. The language used to define the states, operators and preconditions may not be changed and the processes for selection, matching and firing the operators may not be changed as well, but these processes must be general enough to cater to the newly added macro-operators and they should be designed in such a way as to make use of the heuristics generated or refined by the learning process.

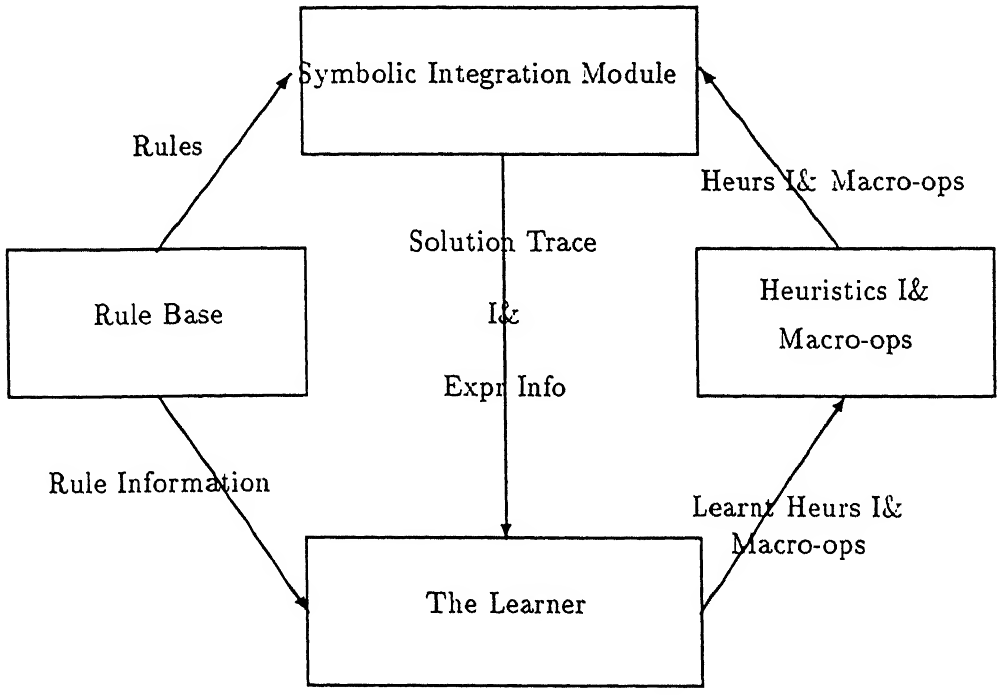

Figure 3.1: The System Model

The two important methods for integration are using by-parts rule and the integration by substitution method. Therefore, we would like to form some heuristics for these methods.

The figure 1 shows the system modules and their relationships. The system contains following modules:

- *Symbolic Integration Module.*
- *Knowledge Base for operators.*
- *Knowledge Base for heuristics and macro-operators.*
- *The Learner Module.*

The specificaitons for these modules are discussed in the following sections.

## 3.3 Symbolic Integration Module

This module is responsible to solve the integration problems and provide the learner with the examples to learn from.

### 3.3.1 Input to SIM

This module takes the information from the following modules:

- Knowledge base for operators

  and

- Knowledge base for heuristics and macro-operators.

The information it requires from the knowledge bases is given a state of the problem situation, which operators, heuristics, and macro-operators are applicable to the problem. This module takes the information from the two knowledge bases. These knowledge bases contain operators, heuristics to guide the usage of these operators and the macro-operators. The information that this module requires from them is given a problem state which operators, heuristics, and macro-operators are applicable to that state. It also asks these two knowledge bases to fire appropriate operators or heuristics or an appropriate macro-operator. The knowledge bases upon such a firing give out the new problem state to this module.

This module is supposed to take the example from the tutor, may be interactively. The turor is a human being who formulates the types of examples, also feeds in the initial rules and operators to the system.

### 3.3.2 Output from SIM

This module as already specified should solve the integrand and provide the solution to the learner. The requirements for the output are very clear: the solution of the problem. The module is supposed to provide the total solution space it is using, alongwith all the status information about all the nodes in the problem space. The SIM is also supposed to store the solution trace, which it should make available to the Learner.

### 3.3.3 Integration process

The module first asks the tutor to provide an example integrand. After taking the integrand, it treats it as the initial state of the problem. Then it starts interacting with the knowledge bases to match and fire the appropriate operators. At every stage, it asks which operators are applicable to the state at hand; then it selects one of the operators or a macro-operator taking the advice of an appropriate heuristics if one is applicable. The kind of heuristics available are: selection heuristics. rejection heuristics and preference heuristics. The selection heuristics suggest some operator to be selected while the rejection heuristics tells which operators should not be applied at that point and the preference heuristic tells if there is a choice. which operators are more beneficial than the others.

After selecting the operator, the module asks the knowledge base containing the operators to fire that operator, supplying it the current state and then taking the new state from the knowledge base and storing it in the solution trace. This loop is continued till the integrand is solved. The module may try all the solution paths available, since that information (trace) will be very crucial to decide the preferences among possible choices to be used the next time. Therefore, arrangements should be made to exhaust the total solution space.

The search process may be a depth-first search with the help of backtracking to traverse through the problem space.

## 3.4 Knowledge Base for Operators

This knowledge base (KB) contains the operators that are defined in the problem formulation. The KB has following parts: storage of operators and procedures that handle the matching, firing, and other features that relate to the storage of these operators. There can be two types of operators: some may be active operators in procedural format and others may be in passive i.e. data format. The active ones are claled as operators and the passive ones are called as rules. The rules require apprpriate methods to handle them.

### 3.4.1 Operator Acquisition

The rules for this module are supplied by the tutor while the operators are coded by the programmer into the module directly. The rules need routines, designed to accept them and store them in a proper format. The rules will have a precedent part and a consequent part. The preconditions are not explicitly mentioned, but they are implicit in the precedent part. Since our system is not supposed to learn new rules or operators, all the rules and the operators are hard-coded.

### 3.4.2 Output of KB

The KB is storing the rules and operators and the other modules require to interact with them, so the KB is supposed to provide them with this information. The interaction with the SIM is already explained in the description of SIM. The other important module, namely the Learner requires to know the information about the rules after it gets the solution trace from the SIM in order to arrive at some associations among the problem status and the typical rule/operator that has got fired.

### 3.4.3 Methods for KB handling

Following methods are required:

- Rule Matching : Upon getting the request from SIM, it should be able to check all the rules that can get applied at that state and should convey it back.
- Rule Firing : After SIM selects a rule for firing, this method should take the current state of the problem and fire the rule to transform the state and should give the new state back.
- Rule Information : The Learner requires to know about the precedent part of the rule to check the associations with the problem state to which that rule is applicable.
  and
- Some minor routines for bookkeeping.

## 3.5 Knowledge Base for heuristics and macro-operators

This KB contains all the heuristics and the macro-operators (macro-ops) generated by the system itself. There are no preloaded heuristics or macro-ops. The system may add or modify a heuristic or a macro-op. This module interacts with the SIM and Learner modules.

The Learner after forming some heuristic or some macro-op, gives it to this module which is responsible for the storage, modificaiton and retrieval of them. Similar to KB for operators, this module also has to have routines to handle matching, firing, acquisition and other bookkeeping methods.

## 3.6 The Learner

The Learner has two major components dealing with two different types of concepts: one deals with the problem of generating the heuristics for the integration methods and the other deals with the formation of the macro-ops improving the system search time and ultimately affecting the overall system performance.

The heuristic generator generates three types of heuristics :

- Selection heuristics
- Rejection heuristics
- Preference heuristics

To generate the heuristics or the macro-ops, the Learner needs to know the complexity of the integral at each node in the solution trace.

### 3.6.1 Complexity of Integral

At this point, we are going to define the notion of complexity and the need to know such a value.

For some integration examples, there may exist more than one solutions, or at least more than one path. We have already made it a point to have all the possible solutions of an example by expanding the choice points in the solution space. We have provided this facility in the SIM module. When such choices are available, the SIM tries to exhaust those possibilities, and gives the solution path for those alternatives.

The heuristic generator will try to generate a preference heuristic which will assign priorities to the involved rules so that next time the SIM can take help of that heuristic to choose the 'better' path. Now, what do we mean by *better*? We human beings have some idea about the complexity of the integrand expression. No doubt, we gather this notion by experience. There is no exact or formal definition of the complexity of the expression but we can vaguely think (rather feel) that one expression is more complex than the other.

Now, if our system has to generate some bias towards the different rules applicable at a certain point, then it needs to know about this complexity notion. Generally, the complexity is inversely proportional to the ease with which we can reduce an expression to the standard table lookup of the rule base. This is still not an easy task, since integration is the reverse process of derivation and we can not certainly find an easy way to classify the integration rules in order to check the closeness of the expression to get the complexity. This task itself can be a good problem to apply learning strategies. In the next chapter, we will define a crude way to get an estimate of the complexity of the expression.

### 3.6.2 Heuristic generator

The solution space is a n-ary tree in which at a particular node one can get all the available rule choices. It also stores the integral expression at that node, so that the solution trace can be made available alongwith the rule choices. All the nodes bear a flag denoting the status of the node. A node can be a Successful one, a candidate one, or a Failed one. The heuristics are generated in the following manner:

1. **Selection heuristics:** The learner traverses the solution space in in-order traversal, and whenever it gets a node which is a successful one, it generates the selection heuristics for that node. It first takes the information about the match point of that integral expression and the information about the precedent part of the rule then it tries to match these two pieces of information in order to decide why the rule got matched at that match-point. These associations form the preconditions of the heuristic which denotes that if we get such a situation in future, then try this rule, it may prove an applicable one.
2. **Rejection Heuristics:** Similar to the selection heuristics, when we get a failure node, we generate the associations and form the rejection heuristic.
3. **Preference Heuristics:** When we encounter a node where we can have the choice of multiple rules being applicable, depending upon the complexity value we decide which rule is preferable. The higher the complexity value difference a rule can bring the better the rule.

### 3.6.3 Constructive Induction

While forming the selection heuristics, we are matching the information of the rule precedent part and the match-point in the expression. Here we take advantage of the constructive induction to form the preconditions of the heuristics. The general the preconditions the better the heuristics. To get maximum generalization, we go on generalizing the class of the expression till we reach the limit set by the precedent part. This generalization is achieved by constructive induction.

### 3.6.4 Macro-operator generator

Macro-ops are generated in order to minimize the search time needed to check which rule can match the expression. When we have certain situation in which, we have already seen a typical sequence of rules getting fired, we can try the same sequence again to arrive at a similar simplified expression. Potentially, after examining each trace, we can take any subsequence of rules and term it as a macro-op. In this way, we may get infinitely many macro-ops and the time required to check them for applicability and other bookkeeping operations will take more time and instead of improving the system performance we may make it to go down.

One more situation that we face is the complexity value may go up temporarily upon application of a rule, which will come down after a few more stages, ultimately solving the example. For example, after the application of the by-parts rule the expression apperantly more complex, but if the inner derivative or integrals are solvable then the overall expression complexity may come down drastically. But, given a choice, the preference heuristics may choose the other choice simply because it reduces the complexity at that point. Hence, next time it will be more difficult for the by-parts rule to get applied. Therefore, we can generate, a macro-op here, which will take the subsequence starting from the point where the complexity value goes up, to the point where it becomes less than the initial value. This way, apparently, the macro-op will reduce the complexity of the expression.

The preconditions for the macro-op can be formed using the information of the overall expression at that stage. This denotes the process of macro-op formation.

# Chapter 4

# The Integration Module

This chapter describes the implementation details for the integration module (SIM). First, we will discuss about the choice of representation of the expression, expression handling routines, the solution space alongwith the expression state, the rulebase and the other knowledge bases.

## 4.1 Language for Implementation

Our system has two main procedural components: The integration module and the learner module and two main knowledge bases containing the rules. heuristics and macro-ops. We also need routines to handle these knowledge bases. The language that we chose for implementation is C++ because of its object oriented features. We can define the knowledge base in terms of objects and we can write methods to handle these objects. The design becomes simpler and more modular. We can start with the bare framework and then go on adding the functionality into it.

We have defined the classes for objects like rule, rulebase, heuristics, macro-op, solution, symbol table, etc. Also we have defined the appropriate methods for these classes. All these definitions are provided in Appendix.

## 4.2 Grammer for Expressions

The grammer defines what can be termed as an expression. It defines which are the terminal symbols and which are the non-terminal symbols. It also defines the types used for operands and functions.

### 4.2.1 Operand/function types

The following types are used to define the operands/functions:

- Constants:

  CONST_VAL: Any integer or real constant.

  CONST_VAR: a, e, m, n, r, k are varibles which can take constant values.

- variables: ID: u, v, w, x, y, z are variables.

- Algebraic operators:

  ALG_OP: +, -, *, /, _(unary minus).

  EXP_F: ∧ (Distinction between exponentiation and power is done semantically).

- Functions:

  TRIG_F: sin, cos, tan, csc, sec, cot.

  LOG_F: log.

- Expression: EXPR: E, E1, E2, ...

- Sign: SIGN: I(integral), D(derivative).

### 4.2.2 Expression hierarchy

The terminals are constants, operators and functions while the non-terminals are variables and expressions. The expression hierarchy is as shown in figure 2. This hierarchy is used to match symbols while we are finding out which rules can match.

## 4.3 Representational Issues

### 4.3.1 Expression representation

The basic issue conerning representations is representing the expressions. This structure is going to be used everywhere in the system. The structure is defined as follows:

```
typedef struct expr
{
```

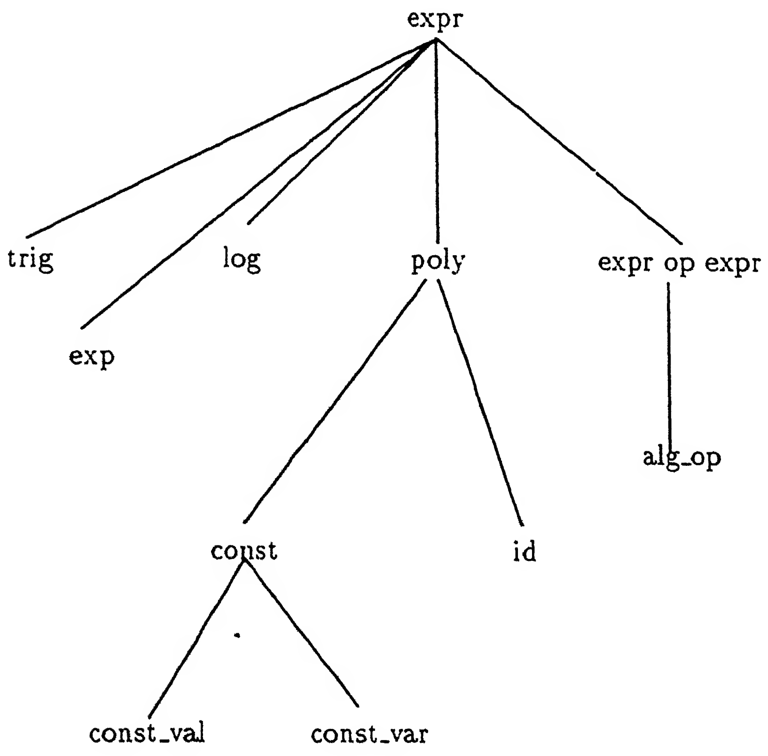

Figure 4.1: The Expression Hierarchy

```
int sign; // Integral / derivative
char op_val[8]; // operator(operand) value
struct expr *left, *right, *parent;
} s_expr;
```

An example is shown in the figure 3 which depicts the representation of $2\ x * x\ 2 \wedge cos * I$. The expression is represented as the binary tree with the node storing the information about the operator value and the sign of the expression alongwith proper pointers to maintain the tree structure.

### 4.3.2 Solution space representation

To represent the solution space we are using a class named soln which contains an n-ary tree whose node contains the following information:

- Information regarding the rule at that node

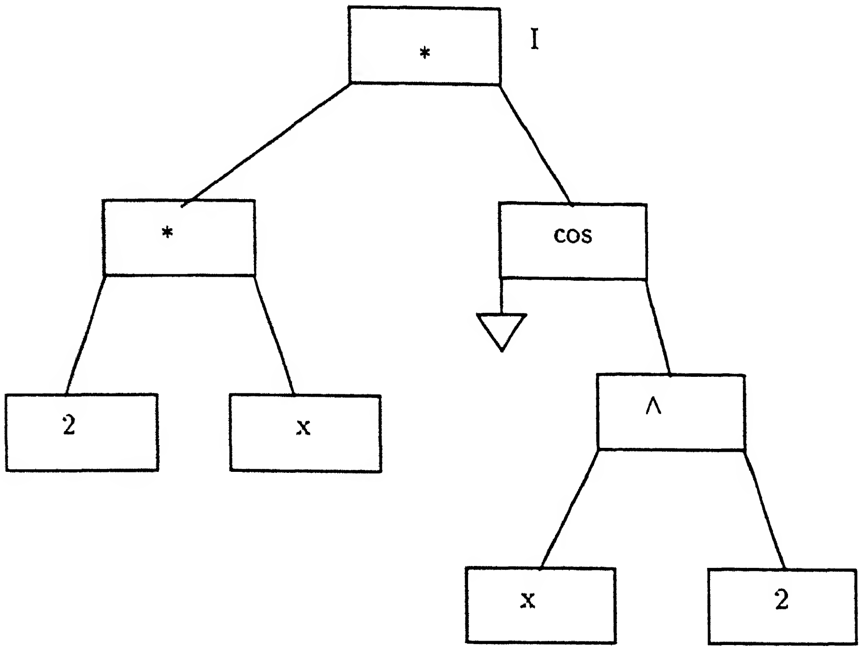

Figure 4.2: Expression Presentation

- Expressions for solution stage and the respective match point.
- Links to parent, child and sibling.

The class also defines many methods that take care of the interface between the `soln` and the other objects or modules. These methods include facilities to add the successor rule choices, adding solution status, printing a solution path, backtracking, and getting various information from the nodes of the solution space.

### 4.3.3 Rule Base representation

We have two classes namely `rulebase` and `rule` defined for this purpose. The `rule` class has got other subclasses such as `int_rule`, `alg_rule`, `der_rule` and `opr_rule` defined for integral rules, algebraic rules, derivative rules and rules concerning the operators. The `rulebase` class contains the instances of all these subclasses and the interaction of all these instances with outside modules is handled through this class.

Every rule has a precedent part and a conseqeuent part; both in the form of expressions. The `rule` class defines methods for setting, matching, firing, printing, and getting informaton about the precedent part of the rule. Since all the interaction is through te rulebase class, that class also has all these methods, only that once such a method is called it checks all the instances for that particular method.

Some operators are defined procedurely. They are:expression simplification, algebraic simplification, constant simplification, division, derivation, identity rules, associativity rules and the substitution operator. Although rules for algbraic simplification, unary minus sign simplification and derivation exist, procedures are required to see if they match and can be fired.

### 4.3.4 Symbol table representation

The symbol table is needed to store the rule unification information at the time of firing the rule. It stores the symbols from the integral expression that get unified with the precedent part of the rule and then they are replaced with their counterparts in the consequent rule to get the transformed expression.

The symbol table contains two types of unifications: since the expression non-terminal can get matched with any type of symbol or another expression and hence, the symbols that get matched with the expression non-terminal symbol are stored seperately and unifications with other non-terminals are stored seperately.

## 4.4 Integration Process

The advantage of using Object-oriented approach becomes evident here. The delegation work to the various classes makes the design clearer and easier to implement. Since, the majority of work is assigned to the methods that deal with their encapsulated data structures. the botheration of other modules becomes less and less.

The basic algorithm for integration problem solving remains simple. Try to get choices for successor rules, operators, heuristics, or macro-operators, select one of them. fire that choice, and continue till you solve the integral. If no successor choices are availble, backtrack till you get any choice point, start from there and again continue.

We are using depth-first search here. After getting a solution, the tutor is asked whether we need to proceed to get other solutions also. In that case if alternatives are available, they are also exhausted. The learning process is called when the solution space is exhausted.

### 4.4.1 Obtaining choices

The algorithm for obtaining choices is as follows:

- First, it is checked whether any operators are applicable or not.
- Then if any selection heuristics are applicable, the rules suggested by them are checked for matching.
- else if any macro-ops are getting matched the first of them is fired.
- else rulebase is checked for matching of any integral rules.
- if any of these are successful, the choices are taken as possible successor candidates and then they are stored in the solution space.
- otherwise, failure is reported so that the main integration routine can try backtracking.

### 4.4.2 Matching

- **Rules**: The rules check whether their precedent part can match anywhere in the given expression. For this they use the expression hierarchy.
- **Heuristics**: The selection heuristics are searched for possible matching. The attributes gathered from the expression are matched with the preconditions of the heuristics and if they match then the rules suggested in that euristics are checked for matching. If any of them match, they become the successor choices.
- **Operators:** The procedural operators are checked for possible firng.
- **macro-ops:** The preconditions of macro-ops are checked in a similar way to that of heuristics but after their preconditins match, the rule sequence in the macro-op is fired. These rules are not checked for individual matching.

After you get the choices, the rejection heuristics are checked to reject any rules as per their suggestions. After this filtering, the preference heuristics are checked to assign the priorities. If no preference heuristics are available then the first choice is selected by the depth-first search mechanism.

### 4.4.3 Firing

The rules are fired in the depth-first search manner. The rule class is given the integral expression. The rule class finds the match-point and then stores the unification information in the symbol table then taking the consequent part of the rule and putting the unification information back the new state of the integral expression is achieved. The *symbol table* is a class and hence the work related to unification is delegated to it.

### 4.4.4 Backtracking

The backtracking problem is delegated to ths soln class. If no successor choices are available then the method backtracks to the parent till it finds some alternative. While backtracking, it asks tutor to suggest any operator that the tutor may find useful at a particular node.

### 4.4.5 Expression handling

The structure used much extensively, is the expression structure. Therefore, many routines are provided to handle the expressions, like, converting string to expr and vice versa, printing an expr, copying exprs, comparing two exprs, freeing the structure, etc. Apart from this, We need some procedural operators to simplify the integral expressions, they are discussed in the next section.

## 4.5 Procedural Operators

The procedural operators are diiscussed below.

### 4.5.1 Simplificaitons

The type of simplifications performed on an integral expression are as follows:

1. *Expr simplification*: In this various operations are done on an expression. The unary minus sign simplification using rules is done if necessary. The terms are arranged in order, which is essential at least for division operator. Also, algebraic simplifiocation is done if required.

2. *Algebraic simplification*: The rules used for algebraic simplification are given in the appendix. e.g. the distribution of * (multiplication) over +- (addition), power rules, etc.
3. *Constant simplification*: The constants are simplified and also identity rules are applied for simplification.
4. *Division*: Currently, only division of two polynomials is supported.
5. *Derivation*: If the expression contains any subexpression that has derivative sign then derivation rules are used there. Since standard rules exist for derivation, the derivative simplification is made rule-based. These rules are also given in the appendix.
6. *Identity rules*: This is the inverse process of simplification where the expressions are expanded using identity rules so that if any integral or algebraic simplification rules may match after such expansion.
7. *Associativity Rules*: Associativity rules are also used in order to see if they can lead to a situation where the integral or algebraic simplification rules may become appicable.
8. *Substitution*: This operator is implemented procedurely, where the system asks the tutor to suggest the proper substitution. Then all the occurances of this expression are substituted by the variable of substitution. The integral is divided by the derivative of the substitution expression, and a flag is set denoting a substitution has taken place. This flag is used at the end to put the substitution back in the expression after the integral is solved.

# Chapter 5

# The Learner

This chapter describes the details of the learning process. The Learner needs to use the information about the expression or at least the match point and the precedent part of the rule so that it can understand why the rule got matched to that expression. The Learner also needs the information about the comlexity of an expression. After knowing about these things and taking the solution space, the Learner can form the necessary heuristics and the macro-ops.

## 5.1 Getting Associations

This module gets the solution space from the SIM and then taking the rule number and class, it gets the information about the precedent part of the rule. Then taking similar information about the match-point it checks for the commonalities and forms the attributes that will form the precondition part of a heuristic.

### 5.1.1 Expression information

The routine `get_expr_info()` takes as its argument a pointer to the expression and then it fills the slots of the structure for expression information defined in the primary definitions' header file. The definition the expr_info structure is:

```
typedef struct comp_inf
{
int comp_type; // component type
int integral; // if comp has an integral sign
int deriv; // if comp has an derivative sign
int expo_type; // exponent type
float comp_deg; // degree of the comp
int fract_deg; // if thet deg is a fractional one
float denr_deg; // denomiator deg if the comp has a div pt
char op[10]; // operator/operand value
int op_type; // its type
char log_base[10]; // base of log function
char exp_base[10]; // base of exp function
int fn1_type; // function1 type
int fn1_op;
int fn2_type; // function2 type
int fn2_op;
struct comp_inf *fn1_inf, *fn2_inf;
} s_comp_inf;

typedef struct exp_info
{
s_comp_inf *c_info[MAX_COMP];
} s_expr_info;
```

The expression is divided in components and then the slots of the structure s_comp_inf are filled by the routine. The routine first decides the type of the components and then depending upon the type it fills the slots.

The possible types are:

- CONST : for constant values and variables.
- ID : for variables or polynomials of degree=1.
- POLY : for polynomials.
- TRIG : for trigonometric functions.

- LOG : for logarithmic functions.
- EXP : for exponential functions.
- MULT : for expressions containing multiplication of two functions.
- EXPR : any other expression or expressions containing dissimilar functions.

The fn1_type is used to denote the operand of the trignometric, log, and exponential functions. It is also used to denote the first function in the MULT and EXPR type of components. In those cases the fn2_type is used to denote the second function.

### 5.1.2 Forming Attributes

The get_assocs function gets the solution space from the SIM, the integral solver. To form the attributes, it uses the information about the match-point for that rule as well as the precedent part of that particular rule. From the information, it gets the attributes of that expression. The possible attributes are:

- Types of expression : CONST, ID, POLY, TRIG, LOG, EXP, MULT and EXPR.
- Types of operand/opeator : CONST_VAL, CONST_VAR, ID, ALG_OP, TRIG_F, LOG_F, EXP_F and EXPR.
- Signs : INTEGRAL, DERIV.
- EXPO_TYPE : denotes exponent type for exp_f.
- COMP_DEG : component degree.
- FRACT_DEG : if that degree is fractional.
- DENR_DEG : denominator degree if division point exists.
- OP : denotes the operand/opeator value.
- EXP_BASE : exponential function base.
- LOG_BASE : log function base.

After we get such attributes for both the expression and the precedent part, a routine checks which of the attributes match. These attributes are then stored in a file "r_match.n" alongwith the rule and the match-point. These matched attributes denote the associations since in actual unification the expressions are matched using the type hierarchy. Here also, we are using the type of the expression as one of the attributes.

The system forms such associations for the integral and the algebraic rules only.

## 5.2 Generating Heuristics

After the associations are formed and stored in an intermediate file, the Learner takes the solution space again and assigns the length of the solution trace information. It also assigns the complexity values to the expressions at each node in the solution space. The complexity value calculations are given in the appendix. The system generates heuristics for selection and rejection of the rules. It also generates the preference heuristics and the macro-ops.

The heuristics are contained in the instances of the classes `s_heur`, `r_heur` and `pf_heur` which are the subclasses of the class `heur_base` denoting selection, rejection and preference heuristics respectively. The definitions of these classes are given in the appendix. The classes also have methods to handle these heuristics. The typical methods used for the heuristics are loading the heur_base, storing it, applying the heuristics and modifying the heur_base when new heuristics are learnt.

The heuristics are taken from a file named "h_base" and the modified heuristics are stored in another file "h_base.n" denoting it as the new file.

### 5.2.1 Selection/Rejection Heuristics

The Learner gets the intermediate file "r_match.n" containing the associations. The learner goes through it and assigns the associations as the preconditions and the heuristic is formed. Depending upon the status of that particular node, whether it is a successful one or a failure one, the associations file contains a tag with that association. This tag is helpful while forming the heuristics. If the tag bears that the rule was successful, then the generated heuristic is added as a selection heuristic otherwise it is added as the rejection heuristic.

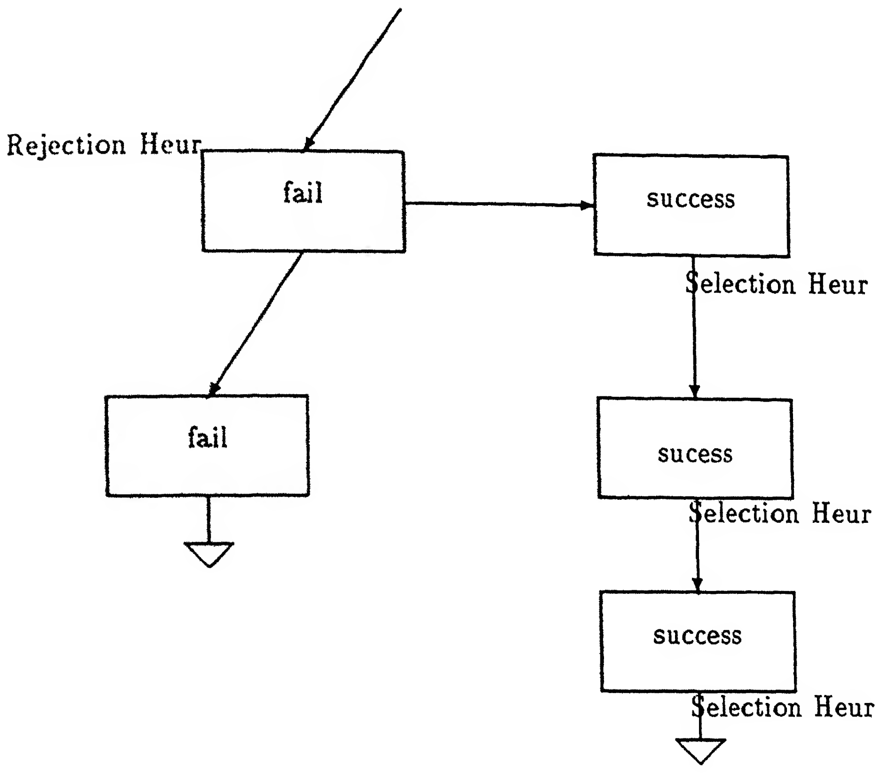

Figure 5.1: Trace for Generating Heuristics

Although, it seems the associations' calculations are similar for both the types of heuristics, there is a subtle difference.

While forming the selection heurstics we use constructive induction and while forming the rejection heuristics we use the strictest conditions posible. This distinction is essential, because, by using the strictest conditions we are lessoning the chances of any rule getting rejected where it could have been useful. The constructive induction helps us to generalize the preconditions so that the rule will be applicable not only to that type of expressions but also some other types of expressions which fall in the same category. We can't use similar technique for rejection heuristics. The next subsection discusses about the constructive induction.

The rejection heuristics are generated for those nodes which are proved failures in the sense that only those nodes carrying the failure tag and they have a sibling which is a successful one. This condition is used for additional surity that we are making rejection heuristics only for those nodes which are useless. Thus, we make the chances of misfiring a rejection rule minimal. The figure 4 shows the situation for generating selection and preference heuristics.

### 5.2.2 Constructive Induction

While generating the preconditions for the heurstics, if the rule at that node is a successful one, the following technique is used. If the type of expression of the precedent part of the rule (*type1*) is hierarchically ancestor than the type of the match-point expression (*type2*) then the type1 is considered for the precondition formation. For example, if the type2 is ID and the type1 is EXPR then EXPR is used to form the precondition. This means that not only the expressions of ID type but also other expressions whose type is hierarchically predecessor to the EXPR can match the preconditions of that selection heuristics. The limit of hierarchical superiority is put using rule information because ultimately, after the selection heuristics suggest that rule, it has to get matched, and hence the limit is practical one.

Such constructive induction is not used for the rejection heuristics for the obvious reasons that it will make that rule nearly unusable because it will get rejected unnecessarily.

### 5.2.3 Preference heuristics

The learner again takes the solution space from the SIM and checks for the nodes where we have got more than one available choices to choose the successor rule. The Learner takes the successful choices for that node. It then forms the attributes for the integral expression state available to that node using the similar information that is available by the expr_info generator.

It also takes the complexity value difference that the respective rules are able to bring forth in the integral expression. The difference is calculated by comparing the complxity values of the integral expression before and after the application that rule. The Learner then sorts the rules in decreasing order of complexity value difference; bringing the rule with maximum difference before in array then other rules. At the time of applying this heuristic, the rules in the choice list are sorted according to order specified by the heuristic, naturally, the rule with maximum complexity value difference comes earlier and because of the depth-first search nature of the SIM algorithm, that rule will become applicable first.

Thus, the preference heuristics assign the priorities to the rules. The figure 5 shows a situation in which a preference heuristic can be generated.

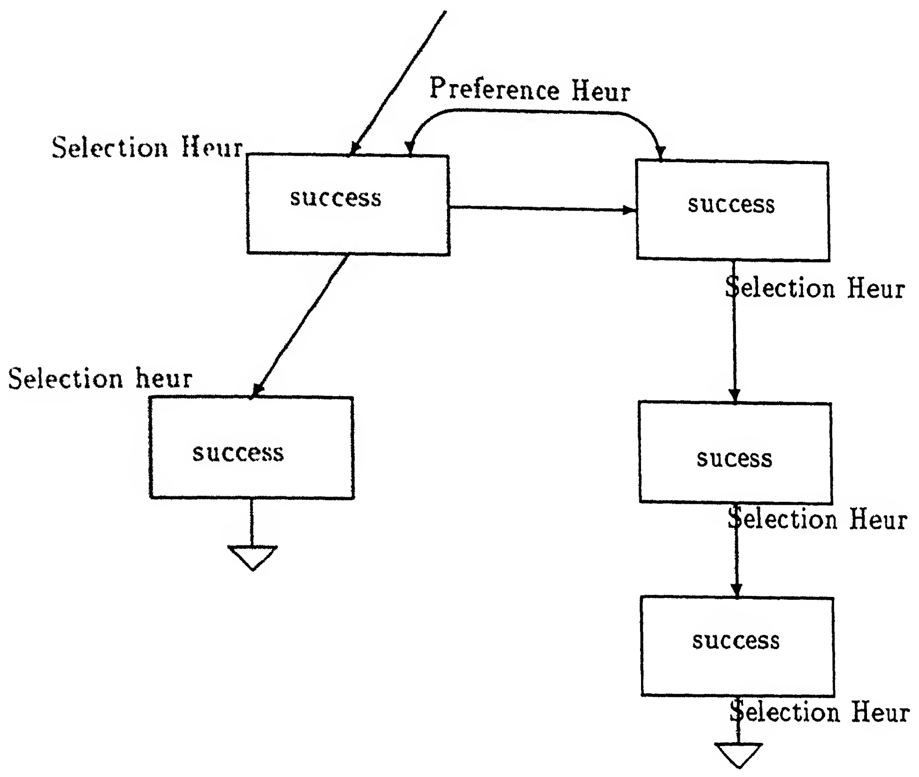

Figure 5.2: Trace for Generating Preference Heuristics

### 5.2.4 Modifying the heur-base

When the new heuristics are generated, they are stored in the existing heur-base. If the preconditions of two heuristics match, they basically denote similar expressions and hence these two heuristics are to combined to form one heuristic. Then the rule list suggested by the heuristic is augmented by the new rule(s). The rules are again sorted in order to decide the which one should get the preference. These sorting is done accoring to the number of times a rule has got fired. For preference heuristics, the sorting is based on the complexity value difference that the rule can presumably bring. if two rules with same complexity value reduction exist then the length of the solution trace is used as a measure to decide which rule provides a shorter solution. If a rule is existing in both the heuristics to be merged then the fire count for that rule is incremented to denote the rule firing frequency.

## 5.3 Macro-opertors

As desribed in the chapter 3, the system also generates the macro-ops using the solution trace. There are a number of ways of generating marro-ops. If we blindly take any sequence, and form a macro-op out of it, we may end up generating infinitely many macro-ops which might degrade the performance of the system rather than improving it. The macro-ops are represented using a class `macrop`. The class `macrop` contains precondition part and the rule-sequence part. The macro-ops are stored in a file "mac.n".

### 5.3.1 Learning macro-ops

The learner takes the solution space from the SIM and then takes all the solution traces from the solution space. All the traces are checked for learning macro-ops.

The Learner takes a trace and checks whether the complexity value of the integral expression is increasing in the trace. Normally, after the application of any rule, the complexity value should either decrease or it should remain constant. If the rule apparently increases the complexity, we would like to put it in the macro-op so that after the sequence suggested by the macro-op is over the complexity of the integral expression would have gone down.

The Learner hence, starts the macro-op rule sequence at such a point and marks the complexity value before that point. The Learner puts all the rules in the trace sequence in the macro-op till it reaches a complexity value of the integral expression less than that value at the start of the sequence. The precondition for such macro-op are taken from the attributes present in the integral expression at the start of the rule sequence.

### 5.3.2 Handling macro-ops

The class `macrop` encapsulates the information about the macro-ops. It contains the methods to load, store the macro-ops from the file, to match and fire them, and to learn them.

A macro-op gets matched if the attributes of the integral expression match with the preconditions of the macro-op. Then the rules from rule-sequence are checked and fired one-by-one. If all the rules get fired then the macro-op is treated successful and the transformed state of the expression is returned back. Otherwise, the macro-op is termed as failed and the original expression is kept the same.

# Chapter 6

# Conclusion

We have seen how our system got developed. At this point we have to study the performance of our system. In general, in inductive learning methods an example sequence is used and the explanation based methods use a single example and then use suitable domain theory to arrive at some concept acquisition. It is very true that even inductive methods do learn something from one example and they do use some domain knowledge to prefer some inductive inferences over the others.

## 6.1 Performance Measurement

The performance of the system can be measured by the number of rule matching that the system does in order to solve a particular problem.

### 6.1.1 Selection and Preference Heuristics

The system is given an example 3 $x$ 2 $\wedge$ $*$ $I$. The system learns some selection heuristics and a preference heuristic using this problem. Then the system is fed another example 2 $x$ $*$ $I$.

Solving first example takes 4223 matchings and it takes 5143 matchings to solve it in two ways.

Solving second example takes 2651 matchings.

After learning the heuristics using first example,

First problem takes just 1250 matchings where the second problem takes 971 matchings.

### 6.1.2 For By-parts rule

The system is given two examples $x\ e\ x\ \wedge\ *\ I$ and $x\ x\ cos\ *\ I$. The systems takes 3785 and 3243 matchings to solve them respectively. After learning the selection heuristics and the macro-op using the first example, the system takes 2897 and 3150 rule matchings respectively.

### 6.1.3 For Integration by Substitution

The system is provided two examples $x\ 2\ \wedge\ e\ x\ 3\ \wedge\ \wedge\ *\ I$ and $x\ e\ x\ 2\ \wedge\ \wedge\ *\ I$. The system takes 7937 matchings to solve the first example and takes 9082 maychings to solve it in two ways. It takes 6893 steps to solve the second example. After learning the selection, preference heuristics and the macro-op, the system solves the examples in 2468 and 2290 matchings respectively. If only macro-op is learnt then the system solves the examples using 4431 and 4253 matchings respectively.

### 6.1.4 Discussion

In the first set the performance improves this much because the heuristics help us at every stage and the two examples are very close in nature which makes the heuristics become applicable. While in the second case, the two examples do not form a very cohesive class. They have only one aspect in common, and that is both of them need By-parts rule and that's where the heuristics help us. Probably a two example sequence specifying one example from By-parts class and another from that particular transcendental class will improve the performance more. In the third set, the performance improves this much because, here also the class is more cohesive and we have all sorts of heuristics and the macro-op up our sleeve.

## 6.2 Comparative Evaluation

[Utg96] is using shift of bias for improving problem solving performance for integration problems in which he emphasizes that the language for describing the states should be explicitly defined, since improving that language will improve the way the states and other concepts are expressed and hence it will improve the overall system performance. He has made the grammer for expressions explicit and his system is able to generate a concept called "Even Integer" required for solving certain trigonometric problems. But in our approach, we have kept this grammer implicit in the system and we want to rather improve upon what is already solvable.

[Mit83] have developed a system called LEX which learns the heuristics for integration by By-parts rule. LEX uses experimentation techniques, in the sense that it generates examples necessary for learning on its own. It has got a heuristics base which stores the partially learnt heuristics and using that heuristics base the Generator generates new examples. The Solver solves those examples and the Critic assigns the instances as positive or negative. Then using that knowledge the heuristics are refined. If you get a positive example then you can generalize that heuristics and if an example proves negative for a particular heuristic, that heuristic gets specialized.

In our system, the system tries to solve the example provided by the tutor and then using that trace and some domain knowledge about the type of expression, it generates some heuristics. If the preconditions of this heuristic match with that of an earlier one, then this rule is simply added to the earlier heuristic and then the rules in that heuristic are sorted again. In our case, this is the only refinement. We want to propose that even after looking at a single example, we should be able to identify a class of problems in which our learnt heuristics would be useful. The same is the case with macro-ops.

From our results, we can say that the problem solving performance does improve after considering a single example. The refinement technique used in LEX specializes in a particular integration method, it tries to build robust heuristics for By-parts rule. In our case, we are generating surface level heuristics. In our view, heuristics are meant as guidelines only, and hence if they are generated in a less expensive method and if they are able to provide an equivalent improvement, then our purpose is achieved.

Our method of generating precondition from the expression attributes, naturally forms classes of integrals such that a particular class of integration rules become applicable to them. This sort of classification helps improving rule-matchings as well as it minimises the generation of new heuristics. This doesn't let the system performance degrade easily with the number of examples studied.

## 6.3 Future Scope for Research

There are many avenues still unaccessed in this problem. We can use various different methods to bring out more improvements in the system performance. We can even take two or more heuristics and then generalize upon them so that we get better heuristics. These heuristics can be used to classify the rules and hence the better the heuristics, the better will be the rule classification. This rule classification will help us improve the retrieval of these rules in an efficient manner. That is here we will be trying classification of instances. One more thing is the heuristics can be generalized using genetic learning methods to get better variations.

The heuristics and macro-ops are not indexed, neither are the rules. We can assign some prirorities to these rules, heuristics and macro-ops after looking at the available solution traces. It will reduce the number of searches in the solution space.

The idea of complexity of an expression is still a vague notion. Since, there can not be any formal method to analyze the complexity, we can make the system learn this complexity aspect depending upon the searches that we had to make, whether successful or futile ones, we can use all this information.

We can apply the techniques of irrelevance theory[Sub90] to organize the domain theory in a better manner. We are already trying to ganerate the macro-ops such that they will act as problem-reduction operators but still we are not generating any concept of that sort. If we make the language for the concept representation explicit then we will be able to generate such a problem-reducing concepts. With this we feel, Symbolic Integration is a good food for thought for applying the learning techniques and a study of this will help the problem solving methods in general.

# Appendix A

# Class Definitions

## A.1 Structure Definitions

```
typedef struct expr
{
int sign;
char op_val[8];
struct expr *left, *right, *parent;
} s_expr;

typedef struct rule_val
{
int rule_no, rule_class;
int val, length;
struct rule_val *next, *prev;
} s_rule_val;

typedef struct trace
{
int rule_no, rule_class;
int val;
} s_trace;

typedef struct space_tree
{
int rule_no, rule_class, status, heur;
int val, length;
char soln_stage[256];
s_expr *match_pt;
struct space_tree *parent, *child, *next;
} s_space_tree;

typedef struct comp_inf
{
int comp_type;
int integral;
int deriv;
int expo_type;
float comp_deg;
int fract_deg;
float denr_deg;
char op[10];
int op_type;
char log_base[10];
char exp_base[10];
int fn1_type;
int fn2_type;
int fn11_type;
struct comp_inf *fn1_inf, *fn2_inf;
} s_comp_inf;

typedef struct exp_info
{
s_comp_inf *c_info[MAX_COMP];
} s_expr_info;

typedef struct ruleno
{
int r_no[MAX_RULES];
int r_class[MAX_RULES];
} s_ruleno;

typedef struct heuristic
{
int precond[MAX_ATTRIB];
int no_prec;
int r_no[MAX_RULES];
int r_class[MAX_RULES];
int val[MAX_RULES];
int length[MAX_RULES];
int r_fire[MAX_RULES];
} s_heuristic;

typedef struct mac
{
int attr[MAX_ATTRIB], n_attr;
int r_no[16], r_class[16];
} s_mac;
```

## A.2 Class Definitions

```
class rule
{
protected : s_expr *precedent, *consequent;
s_expr_info *prec_inf;

public : int match(s_expr *i_expr);
s_expr *get_match_pt (s_expr *i_expr);
int fire(s_expr *i_expr);
int get_fire_count();
int set (s_expr *pre, s_expr *con);
int set_rule_inf ();
s_expr_info *get_rule_inf ();
int print (char *arr);
};

class alg_rule : public rule
{
public : int match(s_expr *i_expr);
s_expr *get_match_pt (s_expr *i_expr);
int fire(s_expr *i_expr);
};

class int_rule : public rule
{
};

class der_rule : public rule
```

```
{
};

class opr_rule : public rule
{
};

class rule_base
{
int_rule *int_rules[MAX_RULES];
alg_rule *alg_rules[MAX_RULES];
der_rule *der_rules[MAX_RULES];
opr_rule *opr_rules[MAX_RULES];
int int_rule_count, alg_rule_count;
int opr_rule_count, der_rule_count;
int match_count;

public :
rule_base ();
int load_rules (int rule_class);
int noof_rules (int rule_class);
int reset_match_count ();
int get_cand_rules (s_expr *i_expr, int rule_class, int cand_set[]);
int get_cands (s_expr *i_expr, int rule_class, int cand_set[]);
int match_rule (s_expr *i_expr, int rule_class);
int match_rule (s_expr *i_expr, s_ruleno *p_rules);
int fire_rule  (s_expr *i_expr, int r_class, int cur_rule);
int print_rule (int r_no, int rule_class, char *buf);
s_expr_info *get_rule_inf (int r_no, int rule_class);
int store ();
};

class soln
{
public :
s_space_tree  *soln_space,  *cur_rule;

soln ();
init_trace (s_expr *i_expr);
int add_succ (s_ruleno *cand, int heur);
int add_choice (char *buf);
int add_soln_stage (s_expr *expr1);
int backtrack (s_expr *i_expr);
int print_soln (s_expr *i_expr);
int get_cur_rule ();
int get_cur_rule_class ();
int get_cur_rule_status ();
int set_cur_rule_status (int stat);
int set_cur_match_pt (s_expr *match_pt);
s_space_tree *get_soln_space ();
int print_orig ();
};

class sym_table
{
char pre_arr[30][10], expr_arr[30][10];
int  conse_arr[30], count, e_count;
s_expr *e_arr[10];

public :
sym_table () {count = 0; e_count = 0;};
int det_val (char *arr1, char *arr2);
int det_val (char *arr1, s_expr *expr1);
int link_val (s_expr *expr1);
int delete_val ();
};

class match_test
{
char pre_arr[30][10], expr_arr[30][10];
int count, e_count;
s_expr *e_arr[10];

public :
match_test () {count=0; e_count=0;};
int test_set (char *arr1, char *arr2);
int test_set (char *arr1, s_expr *expr1);
int reset ();
};

class heur_base
{
protected :
s_heuristic *heur[MAX_HEUR],
int heur_count;

public :
int load_heur (FILE *f_ptr, int flag);
int mod_hbase (int attr[], int r_no, int r_class);
int put_heur (FILE *f_ptr, int i, int flag);
int add_heur (int attr[], int r_no, int r_class);
int add_rule (s_heuristic *heur, int r_no, int r_class);
};

class s_heur : public heur_base
{
public :
s_heur () {heur_count = 0;};
int store_heur (FILE *f_ptr);
int app_heur (s_expr *i_expr, s_ruleno *p_rules);
};

class r_heur : public heur_base
{
public :
r_heur () {heur_count = 0;};
int store_heur (FILE *f_ptr);
int app_heur (s_expr *i_expr, int r_no, int r_class);
};

class pf_heur : public heur_base
{
public :
pf_heur () {heur_count = 0;};
int add_heur (int attr[], s_rule_val *cands);
int add_rule (s_heuristic *heur, s_rule_val *cands);
int mod_hbase (int attr[], s_rule_val *cands);
int store_heur (FILE *f_ptr);
int app_heur (s_expr *i_expr, s_ruleno *p_rules);
};

class macrop
{
s_mac *macro_op[16];
int mac_count;

public :
int learn ();
int form_arrs (s_space_tree *rule_step);
int load ();
int store ();
int match (s_expr *i_expr);
int fire (s_expr *i_expr, int n);
int mac_pr (int r_no);
};
```

# Appendix B

# Rules

The rules are expressed in following format:

precedent expression ⇒ consequent expression

and both these expressions are expressed in post-fix format.

## B.1 Integral Rules

```
a I   =>   a x *
x I   =>   x 2 ^ 2 /
1 x / I   =>   x e log
a x ^ I   =>   a x ^ a e log /
x n ^ I   =>   x n 1 + ^ n 1 + /
E _ I   =>   E I _
E1 E2 + I   =>   E1 I E2 I +
E1 E2 - I   =>   E1 I E2 I -
E1 E2 * I   =>   E1 E2 I * E2 I E1 D * I -
a E * I   =>   a E I *
E a / I   =>   E I a /
a E / I   =>   a 1 E / I *
x e log I   =>   x x e log * x -
x sin I   =>   x cos _
x cos I   =>   x sin
x sec 2 ^ I   =>   x tan
x csc 2 ^ I   =>   x cot _
x sec x tan * I   =>   x sec
x csc x cot * I   =>   x csc
1 1 x 2 ^ - 0.5 ^ / I   =>   x sin 1 _ ^
1 1 x 2 ^ + / I   =>   x tan 1 _ ^
1 x x 2 ^ 1 - 0.5 ^ * / I   =>   x sec 1 _ ^
x cos n ^ I   =>   x cos n 1 - ^ x sin * n / n 1 - n / x cos n 2 - ^ I * +
a x * b + n ^ I   =>   a x * b + n 1 + ^ a n 1 + * /
1 a 2 ^ x 2 ^ - / I   =>   1 2 a * / x a + x a - / e log *
```

## B.2 Algebraic Rules

```
E n ^ m ^   =>   E n m * ^
E n ^ E m ^ *   =>   E n m + ^
m E1 * n E2 * *   =>   m n * E1 * E2 *
m E * n E * +   =>   m n + E *
m E1 * n E1 * E2 + +   =>   m n + E1 * E2 +
x n ^ m *  =>  m x n ^ *
E1 E2 + E3 +  =>  E1 E2 E3 + +
E1 E2 + E3 E4 + +  =>  E1 E2 E3 E4 + + +
E1 E2 * E3 *  =>  E1 E2 E3 * *
E1 E2 * E3 E4 * *  =>  E1 E2 E3 E4 * * *
E1 E2 E3 + *  =>  E1 E2 * E1 E3 * +
E1 E2 E3 - *  =>  E1 E2 * E1 E3 * -
m n E + +  =>  m n + E +
m n E * *  =>  m n * E *
m n E / *  =>  m n * E /
m E n / *  =>  m n / E *
m n E + -  =>  m n - E -
m n E - -  =>  m n - E +
m n E * /  =>  m n / E /
m n E / /  =>  m n / E *
n E + m -  =>  n m - E +
n E - m -  =>  n m - E -
E n + m -  =>  E n m - +
E n - m -   =>  E n m + -
E E log  =>  1
```

## B.3 Derivative Rules

```
E1 E2 + D   =>  E1 D E2 D +
E1 E2 - D   =>  E1 D E2 D -
a E * D   =>  a E D *
E1 E2 * D   =>  E1 E2 D * E1 D E2 * +
a D   =>  0
x D   =>  1
x n ^ D   =>  n x n 1 - ^ *
x sin D   =>  x cos
x cos D   =>  x sin _
x tan D   =>  x sec 2 ^
x sec D   =>  x sec x tan *
x csc D   =>  x csc x cot *
x cot D   =>  x csc 2 ^ _
x e log D   =>  1 x /
a x ^ D   =>  a x ^ a e log *
```

## B.4 Operator Rules

```
E1 _ E2 _ +   =>  E2 E1 + _
E1 _ E2 _ -   =>  E2 E1 -
E1 _ E2 _ *   =>  E1 E2 *
E1 _ E2 _ /   =>  E1 E2 /
E1 _ E2 +   =>  E2 E1 -
E1 _ E2 -   =>  E1 E2 + _
E1 _ E2 *   =>  E1 E2 * _
E1 _ E2 /   =>  E1 E2 / _
E1 E2 _ +   =>  E1 E2 -
E1 E2 - _   =>  E2 E1 -
E1 E2 _ -   =>  E1 E2 +
E1 E2 _ *   =>  E1 E2 * _
E1 E2 _ /   =>  E1 E2 / _
E _ _   =>  E
```

# Appendix C

# Complexity Analysis

The method to get a rough estimate of the complexity value of an expression is as follows:

1. If the expression doesn't contain an integral sign then it's complexity is assumed to be zero.

2. Otherwise, depending upon the type of expression the complexity value is decided for that expression.

The routine assign-val assigns the complexity value to an expression while comp-val assigns value to a component of a polynomial expression similarly mult-val assigns value to a component of the multiple function expression.

```
int assign_val (s_expr *i_expr, int integral)
{
int c_type, type, val=0, i, count;
int val1=0, val2=0, i_flag;
s_expr *match_pt, *ptr;

switch (get_comp_type (i_expr))
{
case CONST :
if (!integral || (integral && (i_expr->sign == INTEGRAL)))
val = 1;
else
val = 0;
break;
case ID    :
if (!integral || (integral && check_int(i_expr)))
switch (get_type (i_expr->op_val))
{
case ID :
val = 1;
break;
case ALG_OP :
val = 2;
break;
}
else
val = 0;
```

```
break;
case POLY  :
i_flag = !(i_expr->sign) && integral;
if (i_expr->op_val[0] == '_')
return (assign_val (i_expr->right, i_flag));
count = get_comp_count (i_expr);
val = 0;
ptr = i_expr;
for (i=0; i<count; i++)
{
if (i == count-1)
{
if (!integral || (integral && check_int(ptr)))
val1 = comp_val (ptr);
}
else
{
if (!integral || (integral && check_int(ptr->left)))
val1 = comp_val (ptr->left);
ptr = ptr->right;
}
if (val1 >val)
val = val1;
}
break;
case TRIG  :
case LOG   :
case EXP   :
i_flag = !(i_expr->sign) && integral;
if (i_expr->op_val[0] == '_')
return (assign_val (i_expr->right, i_flag));
if ((i_expr->op_val[0] == '+')
|| (i_expr->op_val[0] == '-'))
{
val1 = assign_val (i_expr->left, i_flag);
val2 = assign_val (i_expr->right, i_flag);
if (val1 > val2)
val = val1;
else
val = val2;
}
else if (i_expr->op_val[0] == '*')
{
val1 = assign_val (i_expr->left, i_flag);
val2 = assign_val (i_expr->right, i_flag);
if (!val1)
val = val2;
else if (!val2)
val = val1;
else
val = val1 * val2;
}
else if (!integral || (integral && check_int(i_expr)))
{
c_type = get_comp_type (i_expr->right);
type = get_type (i_expr->right->op_val);
if ((c_type == CONST)
|| (c_type == ID) && (type == ID))
val = 3;
else if (c_type == ID)
val = 8;
else
val = 30;
}
```

```
else
val = 0;
break;
case MULT  :
i_flag = !(i_expr->sign) && integral;
val = 0;
if (i_expr->op_val[0] == '_')
return (assign_val (i_expr->right, i_flag));
if ((i_expr->op_val[0] == '+')
|| (i_expr->op_val[0] == '-'))
{
val1 = assign_val (i_expr->left, i_flag);
val2 = assign_val (i_expr->right, i_flag);
if (val1 > val2)
val = val1;
else
val = val2;
}
else if (!integral || (integral && (i_expr->sign == INTEGRAL)))
val = mult_val (i_expr);
else
{
match_pt = get_integral_pt (i_expr);
val = assign_val (match_pt, 0);
}
break;
case EXPR  :
if (i_expr->op_val[0] == '_')
return (assign_val (i_expr->right, integral));
count = get_comp_count (i_expr);
val = 0;
ptr = i_expr;
i_flag = !(i_expr->sign) && integral;
for (i=0; i<count; i++)
{
if (i == count-1)
{
if (!integral || (integral && check_int(ptr)))
val1 = assign_val (ptr, i_flag);
}
else
{
if (!integral || (integral && check_int(ptr->left)))
val1 = assign_val (ptr->left, i_flag);

ptr = ptr->right;
}
if (val1 > val)
val = val1;
}
break;
}
return (val);
}

int comp_val (s_expr *comp)
{
int val1, val2, deg1, deg2;

switch (get_comp_type (comp))
{
case CONST :
return (1);
case ID :
```

```
return (2);
case POLY :
if (comp->op_val[0] == '_')
return (comp_val (comp->right));
if ((comp->op_val[0] == '+')
|| (comp->op_val[0] == '-'))
{
val1 = comp_val (comp->left);
val2 = comp_val (comp->right);
if (val1 > val2)
return (val1);
else
return (val2);
}
if (comp->op_val[0] == '*')
{
val1 = comp_val (comp->left);
val2 = comp_val (comp->right);
if (!val1)
return (val2);
else if (!val2)
return (val1);
else
return (val1 * val2);
}
if (comp->op_val[0] == '/')
switch (get_comp_type (comp->right))
{
case CONST :
return (2);
case ID :
return (2);
case POLY :
deg1 = get_comp_deg (comp->left, POLY);
deg2 = get_comp_deg (comp->right, POLY);
if (deg1 < deg2)
{
if (deg2 > 2)
return (8);
else
return (6);
}
if (deg1 == deg2)
return (3);
if (deg2 > 2)
return (10);
else
return (7);
}
if (comp->op_val[0] == '^')
return (3);
return (comp_val (comp->right));
}
}

int mult_val (s_expr *i_expr)
{
int val, val1, val2, o_deg, i_deg;
int type1, type2, f_type;

if (i_expr->op_val[0] == '_')
return (mult_val (i_expr->right));
if ((i_expr->op_val[0] == '+')
|| (i_expr->op_val[0] == '-'))
{
```

# Appendix D

# An Example

## D.1 Example 1

The integral expression is $x\ 2\ \wedge\ e\ x\ 3\ \wedge\ \wedge\ *\ I$. We have got some selection and preference heuristics and a macro-op from this example.

### D.1.1 Output Trace

scriptsize

```
No of rules 0,25
No of rules 1,25
No of rules 2,15
No of rules 3,14
Enter the expr to be integrated:
orig expr x 2 ^ e x 3 ^ ^ * I
No heurs applicable

The CAND rules :
 matched rule 0,8 at x2^ex3^^*I
Rule no. 0,8 FIRED
Intermediate Stage = x2^ex3^^I*ex3^^Ix2^D*I-

The CAND rules :
 matched rule 2,6 at x2^D
```

```
Rule no. 2,6 FIRED
No heurs applicable
No rules applicable
Intermediate Stage = x2^ex3^^I*ex3^^I2x**I-
No heurs applicable
No rules applicable
* Backtracking
Current rule is 0,8
& current expr = x 2 ^ e x 3 ^ ^ * I
Do you want to give an alternative(y/n)? :Give the choice : Added choice 4,28
* Backtracking to x2^ex3^^*I
The current expr = x2^ex3^^*I
Give the proper substitution u = now the expr is x2^eu^*I
Rule no. 4,28 FIRED
Intermediate Stage = x2^eu^*3x2^*/I
No heurs applicable

The CAND rules :
 matched rule 0,8 at   0.3eu^*I
 matched rule 0,9 at   0.3eu^*I
Rule no. 0,8 FIRED
Intermediate Stage = . 0.3eu^I*eu^I  0.3D*I-

The CAND rules :
 matched rule 2,4 at   0.3D
Rule no. 2,4 FIRED
No heurs applicable

The CAND rules :
 matched rule 0,3 at eu^I
```

```
Rule no. 0,3 FIRED

Intermediate Stage =   0.3eu^eelog/*eu^IO*I-


The CAND rules :

 matched rule 1,24 at eelog

Rule no. 1,24 FIRED


The CAND rules :

 matched rule 1,15 at   0.3eu^1/*

Rule no. 1,15 FIRED

No heurs applicable


The CAND rules :

 matched rule 0,3 at eu^I

Rule no. 0,3 FIRED

Intermediate Stage =   0.3eu^*eu^eelog/0*I-


The CAND rules :

 matched rule 1,24 at eelog

Rule no. 1,24 FIRED

No heurs applicable

No rules applicable

Intermediate Stage =   0.3eu^*

The original integrand and the soln follows :

x 2 ^ e x 3 ^ ^ * I

Rule No : 4,28

SUBST

x 2 ^ e u ^ * 3 x 2 ^ * / I

Rule No : 4,27

DIVISION
```

```
1 3 / e u ^ * I
Rule No : 4,26
CONST_SIM
  0.3 e u ^ * I
Rule No : 0,8
E1 E2 * I  => E1 E2 I * E2 I E1 D * I -
  0.3 e u ^ I * e u ^ I  0.3 D * I -
Rule No : 2,4
a D  => 0
  0.3 e u ^ I * e u ^ I 0 * I -
Rule No : 0,3
a x ^ I  => a x ^ a e log /
  0.3 e u ^ e e log / * e u ^ I 0 * I -
Rule No : 1,24
E E log  => 1
  0.3 e u ^ 1 / * e u ^ I 0 * I -
Rule No : 1,15
m E n / *  => m n / E *
  0.3 1 / e u ^ * e u ^ I 0 * I -
Rule No : 4,26
CONST_SIM
  0.3 e u ^ * e u ^ I 0 * I -
Rule No : 0,3
a x ^ I  => a x ^ a e log /
  0.3 e u ^ * e u ^ e e log / 0 * I -
Rule No : 1,24
E E log  => 1
  0.3 e u ^ * e u ^ 1 / 0 * I -
Rule No : 4,26
CONST_SIM
```

```
 0.3 e u ^ *
The final result is :
 0.3 e x 3 ^ ^ *
Do you want more solns? :starting from  0.3 e u ^ * I
Rule no. 0,9 FIRED
Intermediate Stage = 0.3eu^I*
No heurs applicable

The CAND rules :
 matched rule 0,3 at eu^I
Rule no. 0,3 FIRED
Intermediate Stage = 0.3eu^eelog/*

The CAND rules :
 matched rule 1,24 at eelog
Rule no. 1,24 FIRED

The CAND rules :
 matched rule 1,15 at 0.3eu^1/*
Rule no. 1,15 FIRED
The original integrand and the soln follows :
x 2 ^ e x 3 ^ ^ * I
Rule No : 4,28
SUBST
x 2 ^ e u ^ * 3 x 2 ^ * / I
Rule No : 4,27
DIVISION
1 3 / e u ^ * I
Rule No : 4,26
CONST_SIM
```

```
  0.3 e u ^ * I
Rule No : 0,9
a E * I  => a E I *
0.3 e u ^ I *
Rule No : 0,3
a x ^ I  => a x ^ a e log /
0.3 e u ^ e e log / *
Rule No : 1,24
E E log  => 1
0.3 e u ^ 1 / *
Rule No : 1,15
m E n / *  => m n / E *
0.3 1 / e u ^ *
Rule No : 4,26
CONST_SIM
  0.3 e u ^ *
The final result is :
  0.3 e u ^ *
Do you want more solns? :No more solns
assigning length 12
assigning complexity
stage=   0.3 e u ^ *  val = 0
stage=   0.3 e u ^ * e u ^ 1 / 0 * I -  val = 3
stage=   0.3 e u ^ * e u ^ e e log / 0 * I -  val = 3
stage=   0.3 e u ^ * e u ^ I 0 * I -  val = 3
stage=   0.3 1 / e u ^ * e u ^ I 0 * I -  val = 3
stage=   0.3 e u ^ 1 / * e u ^ I 0 * I -  val = 3
stage=   0.3 e u ^ e e log / * e u ^ I 0 * I -  val = 3
stage=   0.3 e u ^ I * e u ^ I 0 * I -  val = 3
stage=   0.3 e u ^ I * e u ^ I  0.3 D * I -  val = 3
```

```
stage=  0.3 e u ^ *  val = 0

stage= 0.3 1 / e u ^ *  val = 0

stage= 0.3 e u ^ 1 / *  val = 0

stage= 0.3 e u ^ e e log / *  val = 0

stage= 0.3 e u ^ I *  val = 3

stage=  0.3 e u ^ * I  val = 3

stage= 1 3 / e u ^ * I  val = 3

stage= x 2 ^ e u ^ * 3 x 2 ^ * / I  val = 15

stage= x 2 ^ e x 3 ^ ^ * I  val = 8


0. Exit

1. Learn Selection/Rejection Heuristics

2. Learn Preference Heuristics

3. Learn Macro Operator

Give Your choice : learning heurs


0. Exit

1. Learn Selection/Rejection Heuristics

2. Learn Preference Heuristics

3. Learn Macro Operator

Give Your choice :

no of pf heurs 1


0. Exit

1. Learn Selection/Rejection Heuristics

2. Learn Preference Heuristics

3. Learn Macro Operator

Give Your choice : mac_flag ON 4,28 4,27

mac_expr x 2 ^ e x 3 ^ ^ * I

Generated a macro-op 1
```

```
0. Exit

1. Learn Selection/Rejection Heuristics

2. Learn Preference Heuristics

3. Learn Macro Operator

Give Your choice : storing sel heurs 3

storing rej heurs 1

storing pref heurs 1

store macrops 1
```

### D.1.2 Heuristics and Macro-op

```
SELECTION HEUR
LOG OP
24
1
3
SELECTION HEUR
EXP EXP ID EXPO-TYPE COMP-DEG OP EXP-BASE
3
0
3
SELECTION HEUR
EXPR ALG-OP EXPR COMP-DEG OP EXPO-TYPE
15 8 9
1 0 0
2 1 1
REJECTION HEUR
MULT ALG-OP EXP POLY COMP-DEG OP
8
0
1
PREFERENCE HEUR
EXP ALG-OP ID INTEGRAL EXPO-TYPE COMP-DEG OP EXP-BASE
9 8
0 0
1 1
0 0
4 8
MACRO-OP
MULT ALG-OP EXP POLY POLY INTEGRAL COMP-DEG OP
28 27
4 4
```

## D.2 Example 2

Now, we shall use those heuristicss and macro-op generated by first example to solve following:

$x\ e\ x\ 2 \wedge \wedge * I$.

### D.2.1 Output Trace

scriptsize

```
No of rules 0,25 loaded
No of rules 1,25 loaded
No of rules 2,15 loaded
No of rules 3,14 loaded
loading heurs
no of heurs 3 loaded
no of heurs 1 loaded
no of heurs 1 loaded
loading macrops
Enter the expr to be integrated:
orig expr x e x 2 ^ ^ * I
No heurs applicable
Macro-op 0 appicable
firing mac_rule 4,28
Putting substitution u = x2^
Macro-op fired
Intermediate Stage =   0.5eu^*I
selection heur 2 satisfied at   0.5eu^*I
* matched rule 0,8 using heurs at   0.5eu^*I
* matched rule 0,9 using heurs at   0.5eu^*I
pref heur 0 satisfied at   0.5eu^*I
Rule no. 0,9 FIRED
Intermediate Stage =   0.5eu^I*
selection heur 2 satisfied at   0.5eu^I*
selection heur 1 satisfied at eu^I
* matched rule 0,3 using heurs at eu^I
Rule no. 0,3 FIRED
Intermediate Stage =   0.5eu^eelog/*
```

```
The CAND rules :
 matched rule 1,24 at eelog
Rule no. 1,24 FIRED


The CAND rules :
 matched rule 1,15 at   0.5ex2^^1/*
Rule no. 1,15 FIRED
The original integrand and the soln follows :
x e x 2 ^ ^ * I
Rule No : 5,0
  0.5 e u ^ * I
Rule No : 0,9
a E * I  => a E I *
  0.5 e u ^ I *
Rule No : 0,3
a x ^ I  => a x ^ a e log /
  0.5 e u ^ e e log / *
Rule No : 1,24
E E log  => 1
  0.5 e x 2 ^ ^ 1 / *
Rule No : 1,15
m E n / *  => m n / E *
  0.5 1 / e x 2 ^ ^ *
Rule No : 4,26
CONST_SIM
  0.5 e x 2 ^ ^ *
The final result is :
  0.5 e x 2 ^ ^ *
Do you want more solns? :No more solns
```

```
0. Exit

1. Learn Selection/Rejection Heuristics

2. Learn Preference Heuristics

3. Learn Macro Operator

Give Your choice : storing sel heurs 3

storing rej heurs 1

storing pref heurs 1

store macrops 1
```

# Bibliography

[Ban80] banerji,R., *Artificial Intelligence: A Theoretical Approach*, North-Holland, Elsevier Science Pub. Co., 1980.

[Boo89] Booker, L., Goldberg, D., Holland, J., *Classifier Systems and Genetic Algorithms*, Machine Learning: Paradigms and Methods, ed. Carbonell, J., Elsevier Science Pub. Co., MIT Press, 1989.

[Car83] Carbonell, J., *Learning by Analogy: Formulating and generalizing plans from Past Experience*. Machine Learning: An Artificial Intelligence Approach. ed. Michalski, R., et. al., Springer Verlag, 1983.

[Car89] Carbonell, J., *Introduction: Paradigms for Machine Learning*, Machine Learning: Paradigms and Methods, ed. Carbonell, J., Elsevier Science Pub. Co., MIT Press, 1989.

[DeJ90] DeJong, G., Mooney, R., *EBL: An Alternative View*, Readings in Machine Learning, ed. Shavlik J., et. al., Morgan kaufman Pub., 1990.

[Die83] Dietterich, T., Michalski, R., *A Comparative Review of Selected Methods for Learning from Examples*, Machine Learning: An Artificial Intelligence Approach, ed. Michalski, R., et. al., Springer Verlag, 1983.

[Die89] Dietterich, T., *Limitations on Inductive Learning*, Proc of Sixth International Workshop on Machine Learning, ed. Segre, A., Morgan Kaufman Pub., 1989.

[Fla90] Flann, N., Dietterich, T., *A Study of Explanation Based Methods on Inductive Learning*, Readings in Machine Learning, ed. Shavlik J., et. al., Morgan kaufman Pub., 1990.

[Iba85] Iba, G., *Learning by Discovering Macros in Puzzle Solving*, Proc of Ninth IJCAI, 1985.

[Hin89] Hinton, G., *Connectionist Learning Procedures*, Machine Learning: Paradigms and Methods, ed. Carbonell, J., Elsevier Science Pub. Co., MIT Press, 1989.

[Kor80] Korf, R., *Towards a Model of Representational Changes*, Artificial Intelligence, Vol. 14(1), 1980.

[Lan89] Langley, P., *Unifying Themes in Empirical and Explanation Based Learning*, Proc of Sixth International Workshop on Machine Learning, ed. Segre, A., Morgan Kaufman Pub., 1989.

[Mic83] Michalski, R., *A Theory of Inductive Learning*, Machine Learning: An Artificial Intelligence Approach. ed. Michalski, R., et. al., Springer Verlag, 1983.

[Min85] Minton, S., *Selectively Generalizing Plans for Problem Solving*, Proc of Ninth IJCAI, 1985.

[Min89] Minton, S., Carbonell, J., Knoblock, C., Kuokka, D., Etzioni, O., Gil, Y., *EBL: A Problem Solving perspective*, Machine Learning: Paradigms and Methods, ed. Carbonell, J., Elsevier Science Pub. Co., MIT Press, 1989.

[Mit83] Mitchell, T., Utgoff, P., Banerji, R., *Learning by Examples: Acquiring and Refining Problem Solving Heuristics*, Machine Learning: An Artificial Intelligence Approach, ed. Michalski, R., et. al., Springer Verlag, 1983.

[Mit90a] Mitchell, T., *Generalization as Search*, Readings in Machine Learning, ed. Shavlik J., et. al., Morgan kaufman Pub., 1990.

[Mit90b] Mitchell, T., *The Need for Biases in Learning Generalizations*, Readings in Machine Learning, ed. Shavlik J., et. al., Morgan kaufman Pub., 1990.

[Mit90c] Mitchell, T., Kellar, R., Kedar Cabelli, S., *EBG: A Unifying View*, Readings in Machine Learning, ed. Shavlik J., et. al., Morgan kaufman Pub., 1990.

[Rid90] Riddle.P., *Automating Problem Reformulation*, Change of Representation and Inductive Bias, ed. D.P. Benjamin, Kluwer Academic Pub., 1990.

[Sim83] Simon, H., *Why Should Machines Learn?*, Machine Learning: An Artificial Intelligence Approach, ed. Michalski, R., et. al., Springer Verlag, 1983.

[Sim90] Simon, H., Lea, G., *Problem Solving and Rule Induction: A Unified View*, Readings in Machine Learning, ed. Shavlik J., et. al., Morgan kaufman Pub., 1990.

[Stein] Stein, S., *Calculus and Analytical Geometry.*

[Sub89] Subrmanian, D., *Representational Issues in Machine Learning*, Proc of Sixth International Workshop on Machine Learning, ed. Segre, A., Morgan Kaufman Pub., 1989.

[Sub90] Subramanian.D., *A Theory of Justified Reformulation*, Change of Representation and Inductive Bias, ed. D.P. Benjamin, Kluwer Academic Pub., 1990.

[Thomas] Thomas, Finney, *Calculus and Analytical Geometry.*

[Utg86] Utgoff, P., *Shift of Bias for Inductive Concept Learning*, Machine Learning: An Artificial Intelligence Approach, Vol. 2, ed. Michalski, R., et. al., Morgan kaufman Pub., 1986.